INSTITUTO FEDERAL Minas Gerais Campus Formiga	**CURSO DE CIÊNCIA DA COMPUTAÇÃO** Disciplina: Sistema Distribuídos Professor: Everthon Valadão dos Santos

Aluno (a): Alberto Gusmão Cabral Junior
R.A: 0056119

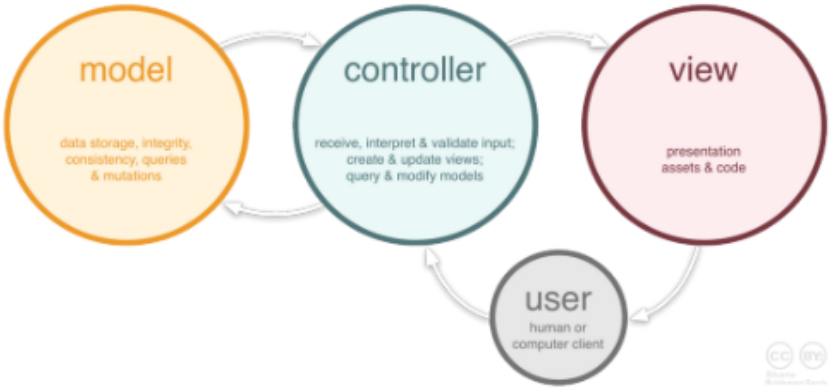
Aluno (a): Gustavo Teixeira de Magalhães
R.A: 0056150

Aluno (a): Henrique Fugie de Macedo
R.A: 0056151

RELATÓRIO DO TRABALHO DE SISTEMAS DISTRIBUÍDOS

1. Parte 1.1 - Arquitetura do Sistema

O trabalho solicitava que fosse projetado um sistema distribuído que executasse um Serviço de Banco Virtual, utilizando como middleware o JGroups com a comunicação entre os membros do cluster. Para a implementação deste projeto, foi utilizada uma arquitetura cliente-servidor por meio do Java RMI, com os membros construídos em três partes diferentes, uma parte que manipula a interação com o usuário, uma parte intermediária que contém a funcionalidade central da aplicação e uma parte que age sobre o banco de dados, no modelo MVC.



Na nossa implementação, o cliente se comunica com o front-end (coordenador) que é o servidor que recebe e executa as requisições, propagando aos demais membros. O Model faz a persistência dos dados no disco, separando de forma até mesmo visual no código, em classes diferentes, a parte que cuida do servidor, da comunicação e controle do cluster, o cliente e o model, que cuida da transferência de estado entre servidores, das transferências feita pelo usuário, tudo que for necessário na persistência de algum dado no disco.

Falando um pouco mais agora sobre a distribuição dos servidores, mais especificamente da arquitetura vertical do sistema, na nossa implementação, todos os membros são réplicas completas do sistema. O método que utilizamos para garantir a consistência entre as réplicas foi um controle de versão, onde a cada operação que o sistema faz, é atualizada a versão que o cluster possui, armazenada de forma persistente no disco, e caso tenha um usuário com uma versão mais avançada, ele replica a versão para os demais clusters, garantindo que todos os servidores, que são controlados por 1 servidor coordenador, tenham a mesma versão do sistema. Em relação a tolerância a falhas, nós utilizamos uma ideia de tudo ou nada, onde se todos os servidores concordam com a transferência, ou operação, ela é mantida e o sistema continua executando normalmente, porém se um deles acusa falha, por exemplo se um dos membros não atualizar, é feito uma expulsão e ressincronismo, caso a maioria falhe, um rollback é realizado.

Por fim, falando sobre a relação de escalabilidade, quando um novo servidor é conectado ao cluster do coordenador, ele solicita o estado e faz a verificação de quem tem a versão mais recente, e com isso, a versão e os dados de todos os membros é atualizado, mantendo a consistência do sistema.

2. Parte 2.1 - Arquivo .xml

O JGroups é uma biblioteca Java que fornece suporte para comunicação confiável entre processos em ambientes distribuídos, oferece também diversos protocolos para lidar com diferentes aspectos da comunicação em cluster.

Nesta seção do relatório, vamos explicar quais protocolos estamos utilizando na biblioteca para a comunicação entre os membros do cluster, dizendo o porquê da escolha dele, e explicando o que ele faz. No arquivo xml do nosso trabalho, existem os seguintes protocolos: UDP, PING, FD_ALL, VERIFY_SUSPECT, NAKACK2, STATE_TRANSFER, CENTRAL_LOCK, GMS e SEQUENCER.

- UDP: o protocolo UDP é um protocolo de transporte sem conexão, que oferece baixa sobrecarga e latência, ele é mais adequado para aplicações onde a entrega deve ser rápida, porém com o UDP ele não garante que cada pacote será recebido. Nós escolhemos UDP pois no sistema distribuído desenvolvido é melhor garantir que as mensagens cheguem, pois caso ela não seja entregue, existem outros protocolos de segurança, ou até mesmo servidores para consultar e corrigir a falha.

- PING: o protocolo PING é usado para descobrir outros membros ativos em um cluster distribuído, ele opera com um nó enviando periodicamente mensagens de “ping” para outros membros esperando resposta, os membros respondem indicando que estão ativos. O motivo da escolha deste protocolo é porque, além dele ser um protocolo que é necessário para o funcionamento do sistema, e também porque é uma verificação mais abrangente, o protocolo à frente é uma forma mais específica de verificar falhas.

- FD_ALL: o protocolo FD_ALL é responsável por monitorar a disponibilidade de todos os membros no cluster para detecção de falhas. Neste protocolo do código, foram feitos alguns ajustes, como o timeout que é o tempo máximo de espera de verificação para considerar o nó como inativo, o intervalo também foi definido para um valor bem baixo, que define a frequência que cada verificação ocorre em milissegundos, por fim também foi alterado o timeout_check_interval que define a

frequência com que é realizada a verificação geral de timeouts. Este protocolo foi utilizado para ter uma verificação de falhas mais precisa em relação ao PING.

- VERIFY_SUSPECT: o protocolo VERIFY_SUSPECT é responsável por verificar suspeitas de falhas entre os membros do cluster, ele trabalha em conjuntos com o FD_ALL por exemplo, para confirmar se um membro suspeito realmente falhou antes de tomar ações definitivas, como a remoção deste membro do cluster. Este protocolo foi escolhido para complementar o FD_ALL e ter mais precisão em situações críticas.

- pbcast.NAKACK2: o protocolo NAKACK2 é responsável por garantir a entrega confiável de mensagens em um ambiente distribuído utilizando confirmações negativas (NAKs) para lidar em situações onde a entrega da mensagem falhou. Este protocolo foi utilizado pois é um dos protocolos necessários para o funcionamento e também para garantir o recebimento das mensagens, pois é uma área crítica que precisa desse cuidado.

- pbcast.STATE_TRANSFER: o protocolo STATE_TRANSFER é utilizado para transferir o estado do grupo entre os membros do cluster, ele é fundamental em situações em que novos membros se juntam ao cluster e precisam obter o estado atual do grupo de membros existentes. O motivo da utilização deste protocolo é que como o sistema do trabalho é dinâmico, no sentido de entrar e sair de membros do cluster, e para manter a integridade e a consistência do sistema, garantindo que todos tenham o mesmo estado.

- CENTRAL_LOCK: o protocolo CENTRAL_LOCK é utilizado para coordenar o acesso a recursos compartilhados no cluster, de maneira que ele fornece uma distribuição de bloqueio distribuído, permitindo que os membros do cluster cooperem para garantir a exclusão mútua em operações críticas. O motivo de estarmos utilizando este protocolo é justamente esse, para que vários processos consigam acessar um recurso, de forma que não gerem arquivo com falhas, ele tranca o recurso para ser acessado ,cada um em sua vez.

- pbcast.GMS: o protocolo GMS é responsável por gerenciar a adesão e saída dos membros do cluster, ele que cuida desta parte dinâmica comentada anteriormente no state transfer e também trabalha em conjunto com o FD_ALL para identificar quem será removido do cluster. Este protocolo foi escolhido pois é necessário para o funcionamentos dos demais e também para controlar de forma precisa quem vai sair ou ser expulso do cluster.

- SEQUENCER: o protocolo SEQUENCER é responsável por garantir que a ordenação de mensagens em um ambiente distribuído, desempenhando um papel crucial quando a ordem em que as mensagens são entregues é importante para a aplicação. O motivo da utilização do sequencer é para termos um multicast totalmente ordenado e também durante alguns testes, foi observado um problema em relação a ordenação das mensagens, como transferências executando na ordem incorreta.

3. Parte 2.2 - MPerf

Para a execução do MPerf no trabalho de sistemas distribuídos, foram executados vários terminais em uma mesma máquina, executando o arquivo “protocolos.xml” como entrada e foram testadas 3 entradas, sendo elas com 10 threads, 1000 mensagens e o diferencial de cada uma, sendo a primeira com 2 terminais, a segunda com 3 terminais e a terceira com 5 terminais.

```
-- sending 1000 msgs
++ sent 100
++ sent 200
++ sent 400
++ sent 500
++ sent 600
++ sent 700
++ sent 800
++ sent 300
++ sent 900
++ sent 1000
-- received 200 msgs (4 ms, 50000 msgs/sec, 50MB/sec)
-- received 400 msgs (5 ms, 40000 msgs/sec, 40MB/sec)
-- received 600 msgs (13 ms, 15384,62 msgs/sec, 15,38MB/sec)
-- received 800 msgs (11 ms, 18181,82 msgs/sec, 18,18MB/sec)
-- received 1000 msgs (10 ms, 20000 msgs/sec, 20MB/sec)
-- received 1200 msgs (3 ms, 66666,67 msgs/sec, 66,67MB/sec)
-- received 1400 msgs (3 ms, 66666,67 msgs/sec, 66,67MB/sec)
-- received 1600 msgs (3 ms, 66666,67 msgs/sec, 66,67MB/sec)
-- received 1800 msgs (2 ms, 100000 msgs/sec, 100MB/sec)

Results:

lar-pc06-4298: 1943 msgs, 1,94MB received, time=48ms, msgs/sec=40479,17, throughput=40,48MB
lar-pc06-23343: 1943 msgs, 1,94MB received, time=55ms, msgs/sec=35327,27, throughput=35,33MB

================================================================================
 Average/node:    1943 msgs, 1,94MB received, time=51ms, msgs/sec=38098,04, throughput=38,1MB
 Average/cluster: 3886 msgs, 3,89MB received, time=51ms, msgs/sec=76196,08, throughput=76,2MB
================================================================================
```

Esta imagem é a execução com 2 terminais.

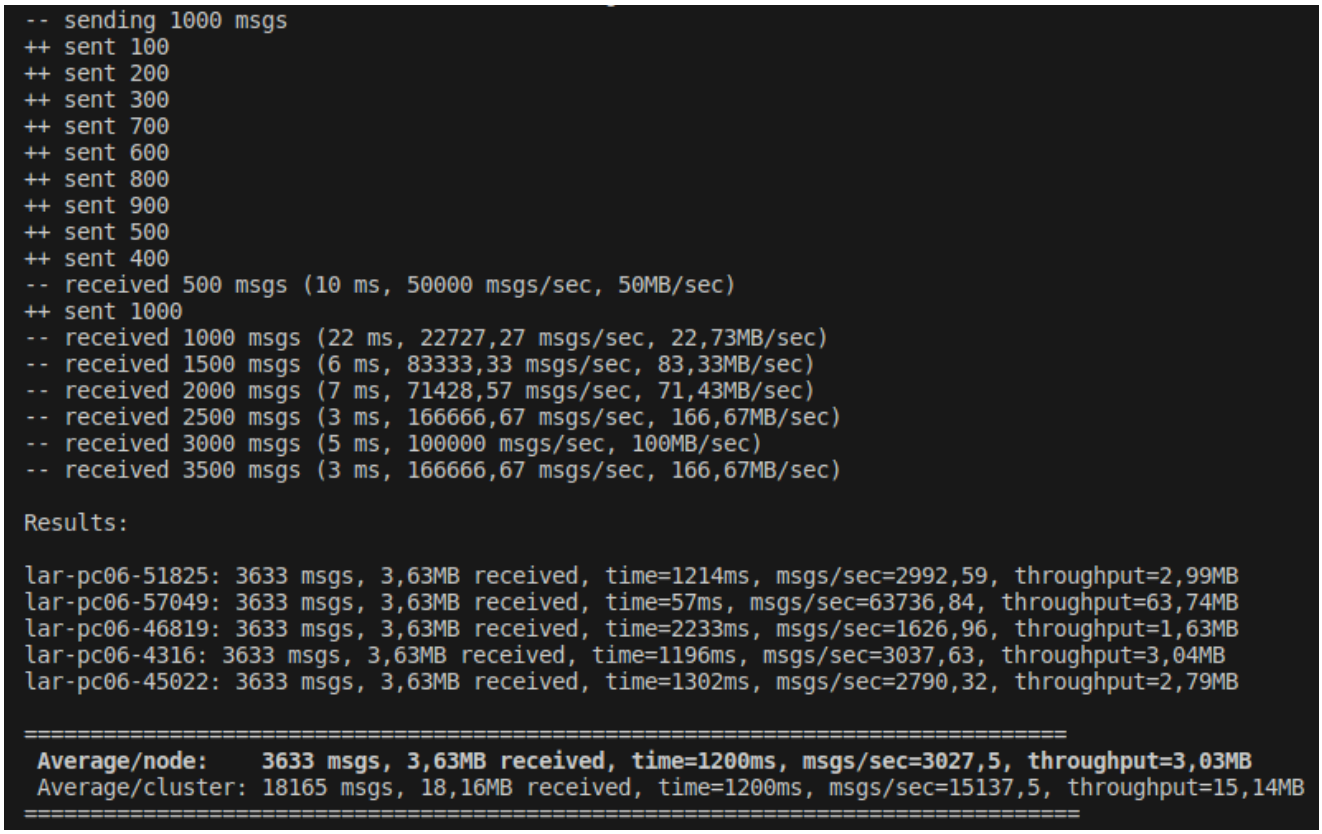
```
-- sending 1000 msgs
-- received 300 msgs (8 ms, 37500 msgs/sec, 37,5MB/sec)
-- received 600 msgs (16 ms, 18750 msgs/sec, 18,75MB/sec)
-- received 900 msgs (13 ms, 23076,92 msgs/sec, 23,08MB/sec)
++ sent 100
++ sent 200
++ sent 300
++ sent 400
++ sent 500
++ sent 600
++ sent 700
++ sent 800
++ sent 900
++ sent 1000
-- received 1200 msgs (37 ms, 8108,11 msgs/sec, 8,11MB/sec)
-- received 1500 msgs (23 ms, 13043,48 msgs/sec, 13,04MB/sec)
-- received 1800 msgs (14 ms, 21428,57 msgs/sec, 21,43MB/sec)
-- received 2100 msgs (24 ms, 12500 msgs/sec, 12,5MB/sec)
-- received 2400 msgs (24 ms, 12500 msgs/sec, 12,5MB/sec)

Results:

lar-pc06-52806: 2641 msgs, 2,64MB received, time=175ms, msgs/sec=15091,43, throughput=15,09MB
lar-pc06-46091: 2641 msgs, 2,64MB received, time=169ms, msgs/sec=15627,22, throughput=15,63MB
lar-pc06-26587: 2641 msgs, 2,64MB received, time=171ms, msgs/sec=15444,44, throughput=15,44MB

================================================================================
 Average/node:    2641 msgs, 2,64MB received, time=171ms, msgs/sec=15444,44, throughput=15,44MB
 Average/cluster: 7923 msgs, 7,92MB received, time=171ms, msgs/sec=46333,33, throughput=46,33MB
================================================================================
```

Esta imagem é a execução com 3 terminais.



Esta imagem é a execução com 5 terminais.

As imagens foram deixadas deste tamanho para manter a qualidade da imagem para ser possível ver os dados adquiridos pelos testes, que mesmo sendo feitos na mesma máquina, apenas utilizando terminais diferentes para serem férias as conexões e teste, o processo demorava um pouco em relação a quantidade de máquinas, tanto que a ideia inicial era tentar com 5, 10 e 15, porém com 10 o programa já não entregava resultados, como observado no laboratório e testado, ele simplesmente não terminava de entregar a saída inteira, travando antes de Results, por isso a quantidade foi drasticamente reduzida.

Algo que foi possível observar com a variação de terminais foi o tempo que cada um levou para executar, onde a adição de apenas mais 1 terminal dobrou o tempo, e a adição de mais dois, totalizando 5, aumentou o tempo para 1200ms.

4. Parte 3.1 - Pontos fortes

No nosso sistema distribuído, ele possui uma interface com o usuário para a execução das funcionalidades da aplicação, cada conta possui um id relacionado a mesma, que não pode ser igual para nenhuma conta, possui verificação na criação de conta de forma que contas com o mesmo nome, exatamente igual, não podem ser criadas, também é possível fazer transferências entre contas bancárias, de forma mutuamente exclusiva como solicitado, através de uma trava mutex, também pode ser feita a consulta do saldo e do extrato que são armazenados de forma persistente no disco, e também é possível consultar a montante do banco.

Agora em relação aos requisitos intermediários, o primeiro tópico foi dito que não era necessário, mas foi feita a divisão vertical MVC, também é feito armazenamento no disco

do estado do sistema, com o controle de versão e os dados em relação a estrato e informações do usuário, ao um servidor reingressar no cluster, devido o controle de versão, caso tenha alguma modificação no sistema ela terá as devidas atualizações e também foi feita um mecanismo de segurança em questão da senha do usuário e autenticação dos dados corretos do mesmo.

Em relação aos requisitos opcionais, foi implementado a execução e conexão de usuários em LAN`s diferentes através do comando bind_addr e utilização do programa radmin para conexão. Foi feito também um wrapper do maven para facilitar a execução e replicação do projeto das máquinas para teste.

5. Parte 3.2 - Pontos fracos

Como pontos negativos, nos encontramos alguns problemas em relação a utilização do radmin para conexão em LAN`s diferentes, algumas vezes ele conectava apenas dois usuario, outro problema é que o logout da conta só funciona pelo menu, caso a janela seja fechada a conta fica trava impossibilitando o login.