UNIVERSIDADE FEDERAL DE SANTA CATARINA - UFSC
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CURSO DE CIÊNCIAS DA COMPUTAÇÃO

INE5418 - Computação Distribuída

Professor Odorico Machado Mendizabal

Relatório T2: Replicação de Atualização Adiada

Mariana Mazzo Heitor (20203088)
Raquel Cristina Schaly Behrens (20100544)

9 de dezembro de 2024

1. Execução

1.1. Execução geral

É necessário ter *python3* instalado na(s) máquina(s).

Para rodar, digite o seguinte comando:

python3 main.py

A seguir, o console permite ao usuário escolher: quais clientes e quais servidores serão instanciados naquela instância do programa, sendo necessário que todos os servidores estejam instanciados em alguma instância para o funcionamento; qual cliente instanciado no terminal irá realizar as transações no servidor; e quais operações a transação irá realizar.

O formato de *input* é realizado da seguinte forma, como especificado na impressão do console:

client_id_1; client_id_2; ...; client_id_n

server_id_1; server_id_2; ...; server_id_n

client_transaction_id; (type_transaction_1, infos_transaction_1); (type_transaction_2, infos_transaction_2); ...; (type_transaction_n, infos_transaction_n)

É possível ativar todos os logs de debug no arquivo *settings.py*, colocando *PRINT_LOGS* como *True*!

Caso *PRINT_LOGS* permaneça como *False*, o console irá exibir apenas a transação que está sendo executada naquele instante, os valores armazenados no servidor após a Fase de Término efetivar a solicitação e o resultado de certificação dessa solicitação, informando se houve um *commit* ou um *abort*.

Após isso, o usuário pode escolher outro cliente e outra sequência de operações de uma transação, passando novamente pelas fases de Execução e Término do servidor.

Para parar a execução do programa, basta digitar “encerrar”.

1.2. Testes

Para rodar os testes, digite o seguinte comando:

python3 tests.py

2. Decisões e Estratégias de Implementação

2.1. Arquivos de Configuração de Nodos

Para a configuração dos nodos foram criados dois arquivos de configuração: *client.txt* e *server.txt*. O primeiro serve para configurar os atributos dos nodos clientes do sistema, enquanto o segundo serve para configurar os dos nodos servidores.

Ambos os arquivos possuem o seguinte formato:

id: *IP*, *host, port*

2.2. Criação e Inicialização dos Nodos Cliente e Servidor

Os nodos são divididos em duas classes diferentes: *ServerNode* e *ClientNode*, ambos sendo classes filhas de *Node*. *ServerNode* representa os servidores e réplicas que irão processar as transações, enquanto *ClientNode* representa o cliente que fará as requisições de diferentes transações aos servidores.

Dessa forma, ao inicializar o programa, o usuário deverá informar quais nodos de cliente e quais nodos de servidor ele deseja criar naquela instância de programa. A forma como isso é feito está detalhado na seção 'Execução' do relatório. Como informado, o algoritmo foi implementado de forma a possibilitar a criação tanto de todos os nodos em um único terminal, quanto de um nodo em cada terminal, dando liberdade ao usuário para definir a quantidade que deseja. Entretanto, para o funcionamento correto, todos os servidores disponíveis deverão estar instanciados antes de rodar o programa.

O programa cria, a partir do arquivo de configuração *server.txt*, um objeto da classe *NodeServer* para todos os servidores do sistema. Isso é feito para que o cliente tenha acesso ao *host* e *port* de todos os servidores do sistema, permitindo a comunicação entre eles. Entretanto, somente os servidores escolhidos pelo usuário serão inicializados na instância do programa. Nesse contexto, 'inicializar' o nodo servidor significa criar threads para rodar a função *server*, que é a responsável por executar os principais funcionamentos do nodo, Em relação ao cliente, é criado apenas o objeto dos clientes escolhidos pelo usuário, configurando seus principais atributos.

2.3. Definição das Transações e Acesso aos Dados do Servidor

As transações são processadas a partir da função *transaction* da classe *ClienteNode*. Dessa forma, para que ela seja executada, o usuário irá informar de *input* pelo terminal o cliente que irá executá-la e quais operações a transação deve conter, terminando com *commit* ou *abort*.

Dessa forma, optamos por definir 'transação' como uma lista de tuplas contendo o tipo de operação a ser realizada e as informações correspondentes a ela - o item, em caso de leitura, e o item e o valor, em caso de escrita -, sendo um parâmetro da função *transaction*. Assim, essa lista é percorrida a fim de passar por todas as operações presentes na transação, processando-as na ordem em que se encontram dentro da lista.

Os dados do servidor e de suas réplicas são armazenados a partir de um dicionário *key-value*, no qual a chave trata-se do item e o valor trata-se de uma tupla com o valor correspondente a esse item e sua versão. Essa estrutura de dados foi escolhida por ter uma boa representação do funcionamento de uma consulta a um banco de dados de forma simplificada, tendo fácil implementação e servindo ao propósito requisitado pelo trabalho. Ela é representada a partir de um atributo *db* da classe *ServerNode*.

2.4. Requisição TCP

Abrir uma requisição TCP envolve dois passos: abrir um 'servidor' para um nodo do programa ouvir requisições e abrir um 'cliente' para o nodo que deseja enviar uma mensagem. Para esse trabalho, escolhemos o protocolo de sockets TCP para implementar as primitivas 1:1 do tipo *send* e *receive*. Nesse contexto, o algoritmo que representa um 'servidor' do TCP é manipulado pelos servidores, enquanto que o algoritmo que representa um 'cliente' do TCP é manipulado pelos clientes.

O algoritmo representando a abertura do 'servidor' é implementado na função *create_tcp_socket* da classe *Node*. Essa função é responsável por criar um socket de protocolo TCP de domínio *AF_INET* e tipo *SOCK_STREAM*. Com isso, ele estará ativo 'ouvindo' os protocolos que chegarem no endereço *{ip:porta}* daquele nodo. Essa ação é executada pela função *handle_tcp_client* da classe *Node*, que é responsável por decodificar e tratar as mensagens recebidas pelos 'clientes', enviando

uma resposta de retorno, sendo ela o valor da consulta de dados de um item solicitado pelo cliente. Para garantir que não haja conflito ao vários clientes tentarem realizar uma leitura de um dado concorrentemente, optamos por utilizar a chamada TCP em um loop *while True*, fazendo com que ele fique o tempo inteiro ouvindo qualquer requisição de leitura que chegue de um cliente a ele.

Em relação ao 'cliente', executado na função *create_tcp_client*, é criado um socket semelhante ao do 'servidor', porém, ao invés de ouvir requisições, ele conecta-se ao 'servidor', a partir do endereço *{ip:porta}* do nodo desejado. Sua função trata-se basicamente de enviar uma mensagem ao nodo servidor solicitando o valor de um determinado item. Ao receber um valor de resposta do 'servidor', esse valor é retornado ao nodo cliente, permitindo a sua utilização. Vale ressaltar que o 'cliente' tentará enviar a mensagem *max_attempts* - variável definida na função para definir a quantidade de tentativas - vezes, imprimido um *log* no console caso falhe. Isso é feito para lidar com um *ConnectionRefused* ao vários clientes, concorrentemente, tentarem realizar uma operação de leitura ao servidor.

2.5. Algoritmo de Transação

O algoritmo da função *transaction* foi implementado seguindo o modelo do pseudo-algoritmo *T* disponibilizado no enunciado do trabalho, com pequenas alterações para encaixar com o resto do código.

Para isso, irá iniciar selecionando um servidor aleatório. Isso é feito sorteando um número com *random* e buscando o servidor que possua esse valor como *id*. Em seguida, será percorrida toda a lista de transações. Caso a operação seja uma escrita, apenas armazena o valor da operação - representado por uma tupla *(item, valor)* na lista de escritas do servidor.

Já, ao encontrar uma transação de leitura, tem duas possibilidades: o item estar entre os que serão sobrescritos ou não. Caso esteja, optamos por 'simular' a leitura imprimindo no terminal o valor correspondente ao último valor de escrita encontrado. Caso contrário, irá tentar fazer uma conexão TCP com o servidor escolhido, retornando o resultado dele, que trata-se de uma consulta aos dados do servidor a partir de uma *key* representada pelo item. Esse resultado de retorno, representado por uma tupla *(item, valor, versão)*, será adicionado na lista de leituras do servidor.

Por fim, ao chegar no final da transação - representado por um *commit* ou

abort -, irá informar o resultado ao usuário. Além disso, caso haja um *commit*, será realizado um broadcast com difusão atômica para o servidor realizar a etapa de Término, verificando se de fato será possível realizar um *commit* ou se irá resultar em um *abort*.

2.6. Broadcast com Difusão Atômica

O broadcast foi implementado na classe *ClientNode*, por meio da função *broadcast*. São feitas requisições UDP do cliente para os servidores, enviando: o *write_server* (lista de escritas), o *read_server* (lista de leituras), as transações, o timestamp (capturado com o horário atual do cliente), para que seja possível estabelecer a ordem total, o id do cliente que está fazendo o broadcast (esse último para que seja possível colocar um log de aviso para informar qual broadcast falhou, caso isso aconteça).

A função *server*, de *ServerNode*, que ficou responsável por receber essa requisição UDP, por meio de *self.handle_udp_client*.

ServerNode possui um atributo *self.message_buffer*, que armazena as mensagens recebidas de broadcasts.

Como o *ServerNode* possui uma thread (declarada em *initialize*) que roda o *handle_udp_client* em loop, que sempre espera uma conexão UDP (ou seja, mensagem de broadcast) e adiciona as mensagens em *self.message_buffer*. Ou seja, *self.message_buffer* está sempre atualizado.

Além disso, *handle_udp_client* também ordena o *Self.message_buffer* pelos timestamps de suas mensagens, garantindo a ordem total, porque mesmo que as mensagens cheguem fora de ordem devido a latência de rede, elas serão ordenadas para que o processamento aconteça na ordem esperada.

O *ServerNode* também possui uma thread (declarada em initialize) que roda a função *server* em loop, ou seja, as mensagens em *self.message_buffer* estão sempre sendo tratadas.

Na função *server* de *ServerNode*, ao tratar uma mensagem, é verificado se o seu timestamp é maior que o timestamp da última mensagem salva. Se sim, ela é processada, e pode resultar em “commit” ou “abort”. Caso contrário, se o timestamp for menor, a mensagem é descartada.

2.7. Algoritmo do Servidor

O algoritmo da função *server* foi implementado seguindo o modelo do pseudo-algoritmo *server* disponibilizado no enunciado do trabalho, com pequenas alterações para encaixar com o resto do código.

Para seu funcionamento, ele é executado por dois tipos de *threads* na classe *ServerNode*: uma responsável por atender requisições TCPs que solicitam a leitura de um determinado dado do servidor e outra responsável por realizar a Fase de Término do sistema, realizando a efetivação. Ambas rodam em um loop *while True*.

Para implementar a parte do algoritmo correspondente a Fase de Término, utilizou-se uma comunicação por broadcast com difusão atômica para requisitar as operações de escrita e leitura salvas pelo servidor, assim como as transações realizadas. Em seguida, é realizada a verificação a respeito das operações de leitura, buscando saber se estão obsoletas. Isso é feito de forma semelhante ao algoritmo apresentado, percorrendo a lista de leituras do servidor e comparando as versões com as armazenadas nos dados do servidor.

De forma semelhante, o tratamento das operações de escrita é feito baseado no algoritmo do enunciado. Isto é, percorre a lista de escritas do servidor e, em seguida, atualiza respectivos valores nos dados armazenados nos servidores, incrementando o número da versão correspondente.

Entretanto, uma diferença entre o algoritmo implementado nesse trabalho e o apresentado no enunciado trata-se de: enquanto o algoritmo do enunciado recebe a mensagem do broadcast dentro de si, esse trabalho criou uma thread que fica escutando por mensagens de broadcast (como explicado na seção de broadcast com difusão atômica), adicionando essas mensagens em *self.message_buffer*. Assim, o server só consome as mensagens adicionadas em *self.message_buffer*, e, ao tratar essas mensagens, executa parte do código do enunciado (linhas 8 até 20 do algoritmo *server)* em *self.process_message*.

Por fim, a função imprime no console a certificação da solicitação de efetivação, informando ao usuário se houve um *commit* ou um *abort*.

3. Casos de teste

Os casos de teste estão no arquivo *tests.json*, cada caso de teste possui um nome, os servidores e os clientes que vão ser executados, os eventos, um parâmetro “ordem” que indica

se cada *conjunto* de mensagens 'evento' deve ser executado em forma paralela.

Os testes podem ser executados pelo arquivo *tests.py*.

Os eventos possuem conjuntos de mensagens. Cada evento possui um "node_id", que representa qual cliente deve executar as mensagens, e as mensagens.

4. Estudo de Caso

É importante destacar que no terceiro caso de teste definido em *tests.json*, testa a **concorrência entre clientes**. Nesse caso de teste é definido que os eventos vão ser executados de forma paralela. Ou seja, o nó 20 vai executar paralelamente dois conjuntos de mensagens, enquanto o nó 21 executa em paralelo um conjunto de mensagens.

Ao executar esse teste, é possível ver que o comando write, x, 4 é enviado antes de write, x, 3. E, de fato, o valor de x fica com valor 3, ao invés de 4, porque seu timestamp foi depois.

É possível ver no teste da próxima imagem que o (write,x,4) acontece no nó 22 primeiramente, mas logo o nó 22 recebe o (write, x, 4) e o valor do x se torna 4. Para esse resultado, é retornado apenas {22: 'commit}.

É possível ver também que o (write,x,3) acontece para todos os nós, resultando em {22: ‘commit’, 24: ‘commit’, e 23: ‘commit’}, e que (write,x,2) não acontece para nenhum, resultando em {23: ‘abort’, 24: ‘abort’, 22: ‘abort’}.

```
Running test: Teste 3 - Vários comandos de vários clientes - Paralelo
('read', 'x', '')
('read', 'x', '')
('read', 'x', '')
('write', 'x', '4')
('write', 'x', '3')
Result of DB 22: {'x': (4, 1), 'y': (0, 0)}
Result of DB 22: {'x': (3, 2), 'y': (0, 0)}
('write', 'x', '2')
Result of DB 23: {'x': (3, 1), 'y': (0, 0)}
{22: 'commit', 24: 'commit', 23: 'commit'}
Result of transaction = commit
Result of DB 24: {'x': (3, 1), 'y': (0, 0)}
Result of DB 23: {'x': (3, 1), 'y': (0, 0)}
Result of DB 22: {'x': (3, 2), 'y': (0, 0)}
{23: 'abort', 24: 'abort', 22: 'abort'}
Result of DB 24: {'x': (3, 1), 'y': (0, 0)}
Result of transaction = abort
{22: 'commit'}
Result of transaction = None
Resultado esperado: commit
Fechando os sockets abertos do caso de teste.
```