Relatório do Trabalho: Implementação do Protocolo DUR (Deferred Update Replication)

Autor: Lucas Mello Muller de Oliveira

Matéria: Computação Distribuída INE

1. Introdução

Este relatório descreve a implementação do protocolo Deferred Update Replication (DUR), um mecanismo de controle de concorrência em sistemas transacionais distribuídos. O objetivo principal é garantir a consistência entre réplicas de banco de dados, permitindo alta disponibilidade e serialização de transações, conforme proposto por [Pedone e Schiper 2012].

A solução desenvolvida inclui:

- Biblioteca de difusão atômica (para ordenação total de commits).
- Implementação do protocolo DUR (controle de versões e teste de certificação).
- Clientes e servidores replicados para simulação de transações concorrentes.
- Casos de teste para validação do protocolo.

2. Implementação

2.1. Biblioteca de Difusão Atômica

Abordagem: Foi utilizado um sequenciador central para ordenar mensagens de commit.

Mecanismo:

O sequenciador (sequencer.py) recebe solicitações de commit, atribui um número de sequência único e as envia para todos os servidores.

Formato das mensagens:

{

```
  "type": "commit",
  "cid": 1,
  "tid": "t1234",
  "rs": [{"item": "x", "version": 0}],
  "ws": [{"item": "x", "value": 15}],
  "sequence": 1
}
```

Tecnologia: Sockets TCP com garantia de ordem FIFO.

2.2. Protocolo DUR

Fases da Transação:

Execução:

- Leituras são realizadas em servidores aleatórios.
- Escritas são armazenadas localmente no write_set.

Término:

- O cliente envia read_set e write_set via difusão atômica.
- Servidores verificam se as versões em read_set estão atualizadas (teste de certificação).
- Se aprovado, o write_set é aplicado e as versões são incrementadas.

Lógica do Servidor:

```
def process_commit(self, commit_request):
    for item in commit_request["rs"]:
```

```
        if self.db[item]["version"] > item["version"]:
            return "abort"
    return "commit"
```

2.3. Componentes Cliente e Servidor

Servidores:

- Réplicas normais: Atualizam o banco de dados após commits válidos (server.py).
- Réplicas com falha: Rejeitam operações para simular cenários de erro (server_falha.py).

Clientes:

- Geram transações com operações de leitura/escrita e enviam commits (client.py).
- Suportam delays para forçar concorrência.

3. Casos de Teste

3.1. Transação Sem Conflito

Objetivo: Validar commits bem-sucedidos quando não há concorrência.

Código:

```
client.execute_transaction([
    {"type": "read", "item": "x"},
    {"type": "write", "item": "x", "value": 15}
])
```

Resultado:

```
[Servidor 5001] COMMIT: Transação t1234 efetivada. Novo estado: {"x": {"value": 15, "version": 1}}
```

3.2. Conflito de Versões

Objetivo: Forçar aborto por leitura de versão obsoleta.

Código:

- Cliente 1 lê x (versão 0) e demora para commitar.

- Cliente 2 atualiza x para 30 e commita primeiro.

Resultado:

[Servidor] ABORT: Transação t5678 abortada (conflito de versões).

3.3. Tolerância a Falhas

Objetivo: Verificar resiliência com servidores defeituosos.

Código:

client = Client(99, [5003], 6001)

Resultado:

[Servidor 5003] ABORT: Transação t9357 abortada (servidor em falha).

4. Conclusão

A implementação do protocolo DUR demonstrou:

- Consistência: Transações concorrentes são serializadas corretamente.

- Disponibilidade: Réplicas operam mesmo com falhas parciais.

- Escalabilidade: O uso de um sequenciador simplifica a difusão atômica.

Referências:

- Pedone, F.; Schiper, N. (2012). Byzantine fault-tolerant deferred update replication.

- Mendizabal, O. M.; Dotti, F. L. (2013). Model checking the deferred update replication protocol.

- Bernstein, P. A. et al. (1987). Concurrency Control and Recovery in Database Systems.

Anexos:

Código-fonte disponível em: https://github.com/LucasMe110/deferred-update-replication

Instruções de execução no README.md.