Universidade Federal de Santa Catarina – UFSC
Departamento em Informática e Estatística
INE 410130 – Computação Distribuída
Semestre 2025/1

Definição do Trabalho: Controle de Concorrência em Transações com Replicação de Atualização Adiada (Deferred Update Replication)

Sistemas transacionais visam aumentar a concorrência em sistemas de gerenciamento de dados, enquanto oferecem garantias de consistência e alta disponibilidade. Usando protocolos para controle de concorrência, pode-se estabelecer o nível de consistência desejável, garantindo que transações efetivadas não violem o critério de consistência estipulado (ex. serializabilidade, *snapshot isolation*, etc.). A replicação surge como uma estratégia para aumentar a disponibilidade e vazão do serviço.

Neste trabalho será implementado o protocolo *Replicação de Atualização Adiada - DUR (Deferred Update Replication)* [Pedone e Schiper 2012]. O protocolo DUR reduz a sobrecarga de sincronização permitindo que as ***atualizações sejam confirmadas localmente primeiro*** e ***propagadas para réplicas posteriormente*** de forma deferida. Os exemplos e algoritmos apresentados foram extraídos de [Mendizabal et al. 2013].

Transações são executadas por clientes e servidores replicados são responsáveis por atualizar e manter o estado do banco de dados consistente. A Figura 1 descreve a execução de uma transação. Um ciclo de vida de transação é separado em duas fases: *execução* e *término*. A Fase de Execução abrange todas as operações de leitura e escrita, enquanto a Fase de Terminação certifica uma solicitação de efetivação.

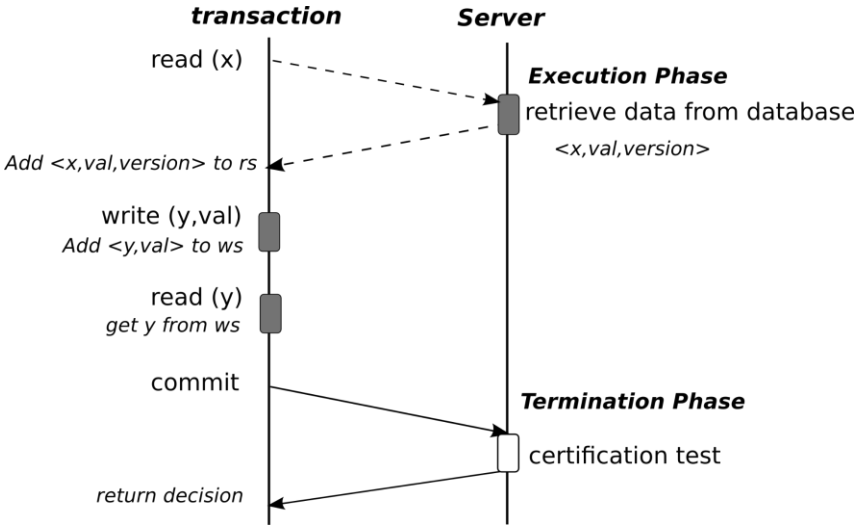

Figura 1: Fases das transações

Cada transação mantém um conjunto de leitura e escrita (*read* e *write sets*). O conjunto de escrita (`ws`) é um conjunto de tuplas com `<item, valor>` e o conjunto de leitura (`rs`) é um conjunto de tuplas com `<item, valor, versão>`.

As operações de escrita são executadas localmente pela transação. Cada item atualizado é mantido pelo cliente no `ws` até que a transação entre na fase de término. Se uma operação de leitura acessa um item já no `ws`, o valor dos dados é copiado diretamente do `ws`. Caso contrário, se o item de leitura não estiver no `ws`, a operação de leitura solicita o valor do item de uma réplica do servidor.

Após executar todas as operações de leitura e escrita, a Fase de Terminação inicia, enviando uma solicitação de confirmação para todos os servidores replicados. Além de conter identificadores de cliente e transação, a solicitação de confirmação também propaga o `rs` e o `ws`. Essas informações são usadas pelas réplicas no teste de certificação. Se o `rs` da transação contiver informações obsoletas, o servidor decide abortar a transação.

Diferentemente das mensagens enviadas durante a Fase de Execução, a solicitação de confirmação é enviada por meio do *protocolo de difusão atômica* (linhas sólidas na Figura 1). Isso é necessário para forçar que todas as réplicas recebam as solicitações de término das transações em andamento na mesma ordem. Uma vez que todos os servidores estejam totalmente replicados e recebam mensagens de confirmação na mesma ordem, as réplicas tomarão as mesmas decisões na mesma ordem, efetivando ou abortando transações.

Algoritmos

Os algoritmos 3 e 4 são descrições de alto nível de processos de transação e servidor. As mensagens que passam por canais comuns são representadas pelos operadores ! e ?. As mensagens de difusão atômica são enviadas pelo bloco de construção `abcast`.

Antes de executar as operações, um servidor é selecionado aleatoriamente (l. 2 do Algoritmo 3). As operações de escrita não exigem comunicação com o servidor inicialmente. Em vez disso, elas são armazenadas no conjunto de escritas (ws) (l. 3-4). Se uma operação de leitura acessar um item de dados atualizado anteriormente pela transação atual, o valor dos dados é recuperado de ws (l. 7-8). Caso contrário, uma solicitação de leitura é enviada ao servidor selecionado e o valor recebido é adicionado ao conjunto de leitura (rs) (l. 10-12). Quando não há operações adicionais de leitura ou gravação para executar, a transação solicita confirmação ou aborto (l. 14 e 18). Além disso, uma mensagem de solicitação de efetivação é enviada a todos os servidores replicados por transmissão atômica (l. 15). Essa abordagem otimista evita várias rodadas de comunicação durante a execução da transação.

O lado do servidor aguarda por mensagens de solicitação de leitura ou solicitação de efetivação (l. 4 e 6 do Algoritmo 4). Ao receber uma solicitação de leitura, o servidor recupera seu valor e versão para o item de dados solicitado (l. 5). Ao receber uma solicitação de efetivação, um teste de certificação verifica se as transações em andamento garantem a serialização [Bernstein et al. 1987]. O servidor verifica se o conjunto de leitura da transação contém itens obsoletos, comparando a versão de cada item de `rs` com a versão do respectivo item no banco de dados. Se pelo menos um item recebido estiver desatualizado, a transação deve ser abortada (l. 8-12). Caso contrário, o servidor decide efetivar a transação. Isso consiste em atualizar a versão local do item no banco de dados, executar todas as atualizações de acordo com `ws` e enviar um resultado de confirmação para a transação solicitante (l. 15-20). Obs. No Algoritmo 4, são inicializadas variáveis x e y (l. 2), apenas. Esta

é uma simplificação para manter um espaço de variáveis pequeno, para efeitos de discussão e verificação. O espaço de variáveis pode ser maior ou mesmo dinâmico, com novas variáveis sendo criadas quando na primeira escrita.

Algorithm 3 T(cid, t)

```
1:  ws ← ∅; rs ← ∅; i ← 0;
2:  choose randomly one of the replica servers s
3:  while t.getOp(i) ≠ commit ∧ t.getOp(i) ≠ abort do
4:      if t.getOp(i) = write then
5:          ws ← ws ∪ (t.getItem(i), t.getValue(i))
6:      if t.getOp(i) = read then
7:          if t.getItem(i) ∈ ws then
8:              return v, s.t. (t.getItem(i), v) ∈ ws
9:          else
10:             c2s[s]!read, t.getItem(i), cid
11:             s2c[cid]?v, version from s
12:             rs ← rs ∪ (t.getItem(i), v, version)
13:     i++;
14: if t.getOp(i) = commit then
15:     abcast.send(com_req, cid, t.id, rs, ws)
16:     s2c[cid]?outcome, s
17:     t.result = outcome
18: else
19:     t.result = abort
```

Algorithm 4 Server(id)

```
1:  lastCommitted ← 0
2:  db[id].setVersion(x, 0); db[id].setVersion(y, 0)
3:  while true do
4:      :: c2s[id]?read, item, cid →
5:      s2c[cid]!db[id].getVal(item), db[id].getVersion(item)
6:      :: abcast.deliver(com_req, cid, t.id, rs, ws) →
7:      i ← 0; j ← 0; abort ← false
8:      while rs[i].getItem() ≠ ∅ do
9:          if db[id].getVersion(rs[i].getItem()) > rs[i].getVersion() then
10:             s2c[cid]!abort, t.id
11:             abort ← true
12:             break
13:         i++
14:     if ¬abort then
15:         lastCommitted++
16:         while ws[j].getItem() ≠ ∅ do
17:             db[id].addVersion(ws[j].getItem())
18:             db[id].setItem(item, ws[j].getVal())
19:             j++
20:         s2c[cid]!commit, t.id
```

Implementação

Deverá ser implementado o protocolo DUR, com base nos algoritmos apresentados e literatura complementar, se necessário. Para a comunicação, serão utilizadas primitivas 1:1, do tipo `send(m)` e `receive(m)` e 1:n, do tipo `broadcast(m)` e `deliver(m)`.

Para as primitivas 1:1, podem ser utilizados *sockets* TCP, que garantirão a entrega e ordem FIFO. Para a comunicação 1:n, será necessário implementar algum algoritmo de difusão atômica. Para este trabalho, pode-se assumir que os nodos não irão falhar, então mesmo uma implementação ingênua de difusão deve ser suficiente, por exemplo o *Best-effort Broadcast (BEB)* [Guerraoui et al. 2011], abordagens baseadas em um nodo sequenciador de mensagens, etc. [Défago et al. 2004].

Devem ser implementados:
1. Os servidores gerenciadores de dados (pode ser um *key-value store*, por exemplo), que podem ser replicados;
2. Clientes concorrentes, responsáveis por produzir transações que operam sobre os dados gerenciados pelos servidores.
3. Casos de teste para validar o protocolos. Elabore situações de concorrência entre clientes em que transações serão efetivadas com sucesso e abortadas (dica: você pode adicionar tempos de espera entre execuções de operações das transações, de modo a forçar situações de interesse).

O formato das mensagens, configurações, etc. ficam à seu critério.

Entrega

O trabalho consiste em:

1. Implementar o programa, incluindo (a) biblioteca para difusão atômica, (ii) o DUR e (iii) partes cliente e servidor para teste.

O trabalho será apresentado em laboratório. O código-fonte e relatório devem ser enviados pelo Moodle para análise e avaliação.

Referências

Attiya, H., Welch, J.. Distributed Computing Fundamentals, simulations, and Advanced Topics. 2004.

Bernstein, P. A., Hadzilacos, V., and Goodman, N. (1987). Concurrency Control and Recovery in Database Systems. Addison-Wesley.

Guerraoui, R., Rodrigues. L. Introduction to Reliable Distributed Programming. 2011

Mendizabal, O. M., Dotti F. L.. Model checking the deferred update replication protocol. *Anais do 31 Simpósio Brasileiro de Redes de Computadores, 2013*.

Pedone, F. and Schiper, N. (2012). Byzantine fault-tolerant deferred update replication. Journal of the Brazilian Computer Society, 18.

Défago, X., Schiper, A., & Urbán, P. (2004). Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys (CSUR)*, *36*(4), 372-421.