Relatório Sistemas Distribuídos

5 de abril de 2025 Grupo A28: ist1105901, ist1106326, ist1106827 Professor: Diogo Pacheco

Introdução

1. Com que nível de dificuldade fizeram o projecto? I am Death Incarnate

2. Que partes das seguintes entregas não foram realizadas ou têm limitações? A1. (cliente servidor sem front-end) A2. (cliente-> front-end -> servidor) B1. (read e put com servidores replicados) B2. (take) C1. (take com expressões regulares) C2. (optimizações de desempenho do algoritmo Xu-Liskov)

Todas as partes foram realizadas.

Implementação

3. Respondam ainda às seguintes perguntas sobre as diferentes fases da vossa solução:

B2. (take) B2.i. De que forma é que o take suporta operações concorrentes e os acessos concorrentes aos tuplos são geridos?

1. **Bloqueio por cliente:** Cada tuplo tem um flag `locked` e um atributo `lockedByClientID`. Quando um tuplo é reservado na Fase 1, fica bloqueado para o cliente específico.
2. **Sincronização:** Todos os métodos no `ServerState` são `synchronized`, garantindo acesso atômico às estruturas de dados; Uso de `wait()` e `notifyAll()` para coordenar threads quando não há tuplos disponíveis.
3. **Rollback automático:** Se um cliente encontrar um tuplo já bloqueado na Fase 1, todos os seus bloqueios são libertados (na Fase 2).
4. **Seleção determinística:** O cliente seleciona sempre o mesmo tuplo (o primeiro da lista) em todas as réplicas, o que garante consistência entre as réplicas.

C1. (take com expressões regulares) C1.i. De que forma é que as expressões regulares alteram o funcionamento do take? Descrevam textualmente o algoritmo implementado com apontadores para o código da submissão.

As expressões regulares modificam o funcionamento da operação `take` ao:

1. **Substituir comparação exata por matching de padrões**: Em vez de procurarmos por um tuplo específico (*string*), procuramos um tuplo que corresponda ao padrão regular (*regex*) fornecido.
2. **Permitir buscas flexíveis**: O cliente pode especificar padrões parciais ou complexos usando a sintaxe Java de expressões regulares.
3. **Aumentar o conjunto de resultados possíveis**: Uma única expressão regular pode corresponder a múltiplos tuplos no espaço de tuplos.

Algoritmo implementado:

1. Cliente envia um pedido `take` contendo um padrão em formato de expressão regular (*regex*), e o seu identificador único (`clientId`).
2. **Processamento no Frontend**: O Frontend determina o *conjunto de votação* (`voterSet`) baseado no `clientId`. Prepara o pedido `takePhase1` com o padrão regex (`searchPatern`).
3. **Fase 1** - Obtenção de candidatos: O Frontend envia o pedido `takePhase1` às réplicas do conjunto de votação. Cada réplica executa localmente: Percorre a lista de tuplos; Verifica cada tuplo usado (`tupleEntry.getTuple().matches(searchPatern)`); Bloqueia todos os tuplos correspondentes para esse cliente (`tupleEntry.setLock(true)`; `tupleEntry.setLockByClientID(clientId)`); Retorna a lista de tuplos correspondentes encontrados.
4. **Resultados**: O Frontend recebe as respostas de todas as réplicas. Calcula a **interseção** das listas recebidas (`List<String> intersection`). Do conjunto de interseção, escolhe o primeiro tuplo.
5. **Fase 2** - Confirmação e remoção: O Frontend envia um pedido `takePhase2` a *todas* as réplicas contendo o tuplo a remover e o `clientId` para validação. Cada réplica verifica se o tuplo está bloqueado

por esse cliente, remove o tuplo do espaço de tuplos (iterator.remove()) e liberta quaisquer bloqueios associados.

6. **Resposta ao cliente**: O Frontend devolve ao cliente o tuplo removido garantindo que o tuplo foi removido de todas as réplicas de forma consistente.

Conclusões:

- **Fase 1** usa matches() para encontrar todos os tuplos que dão match com o padrão regex;
- **Fase 2** usa equals() para operações exatas no tuplo específico selecionado;
- O bloqueio é feito em todos os tuplos correspondentes na Fase 1;
- É removido apenas o tuplo específico que foi selecionado na Fase 2.

C2. (Optimizações de desempenho do algoritmo Xu-Liskov) C2.i. Como implementaram as optimizações de desempenho propostas no artigo Xu-Liskov?

1. Passamos o take para apenas **duas fases**: Fase 1: Entrada na secção crítica. Fase 2. Remover o tuplo e sair da secção crítica.
2. Retornar ao cliente sem esperar por resposta do put: No método processPutRequest, a resposta é enviada antes de esperar pelas confirmações.
3. Retornar ao cliente resposta do take assim que começa a Fase 2: No método processTakeRequest assim que a Fase 1 é concluída (com a seleção de um tuplo), o Frontend pode responder ao cliente e iniciar a Fase 2 em background. O cliente não precisa esperar pela conclusão da Fase 2 (remoção em todas as réplicas), reduzindo a latência.

C2.ii. Como mitigaram os problemas que essas optimizações podem causar?

1. Assegurar que cada réplica executa as operações do mesmo clientId pela ordem de invocação.

O singleThreadExecutor no Frontend (FrontEndService) processa pedidos **um de cada vez**. O que garante, que operações do mesmo cliente, são processadas **na ordem de chegada**, mesmo com retornos antecipados.

2. Quando um cliente invoca put, esperar até que após o último take feito pelo mesmo cliente ter recebido as respostas todas (da Fase 2). Só então é que os pedidos de put são enviados às réplicas.

O singleThreadExecutor serializa put e take, evitando sobreposição. Na **Fase 2 do take**, a remoção do tuplo é atómica (via synchronized no ServerState). Como os puts são processados na mesma thread, **nunca** ocorrem durante um take incompleto.

Testes

4. Quais os testes incluídos que ilustram os aspectos adicionais das vossas soluções? Pasta chamada testes-finais. Que funcionalidades ilustram com esses testes e qual o comportamento expectável?

Temos 2 pastas: **A28** (nossos testes); Teachers (testes feitos pelos professores com pequenas mudanças).
Dentro da pasta A28: **Test1:** Demonstra que operações concorrentes são tratadas corretamente (O take do Cliente1 deve remover <a,1> antes que o Cliente2 possa lê-lo); **Test2:** Testa a operação take com padrões regex (O take com regex deve remover ou <x,apple,1> ou <x,orange,3>).
Para além disso, corremos todos os testes usando time de forma a comparar a **otimização do código** entre a segunda e a terceira entrega, obtendo os tempos médios de 35.309s e 31.483s, respetivamente.

Conclusão

5. Outras notas e comentários

O projeto permitiu-nos explorar os **desafios de sistemas distribuídos**, como concorrência e consistência, com implementações desde operações básicas, até take com regex, algoritmo Maekawa, linearizabilidade e otimizações Xu-Liskov. A maior dificuldade foi gerir bloqueios concorrentes. A principal conclusão foi a importância de sincronização cuidadosa em ambientes distribuídos.