Sistemas Distribuídos – Trabalho Prático

Base de dados concorrente com acesso remoto

Carolina Sofia Lopes Queirós Pereira	A100836
Diogo Luís Barros Costa	A100751
Humberto Gil Azevedo Sampaio Gomes	A104348
Sara Azevedo Lopes	A104179

28 de dezembro de 2024

Resumo

O trabalho prático proposto consiste no desenvolvimento de uma base de dados que armazena, em memória, associações chave-valor. A base de dados é gerida por um servidor, que serve pedidos de clientes comunicados via TCP/IP, garantido a atomicidade das suas execuções. Desenvolveu-se uma solução concorrente, utilizando primitivas de sincronização baseadas em monitores. Procurou-se diminuir a contenção e minimizar o número de *threads* acordadas, pelo que se implementaram diversas estratégias de controlo de concorrência, procurando-se aumentar o desempenho do *software* sem sacrificar a sua correção. O desempenho destas várias estratégias foi medido com recurso a uma ferramenta de testes também desenvolvida, e os resultados dos *benchmarks* realizados apresentam-se neste relatório.

1 Arquitetura do programa

O *software* desenvolvido foi dividido em componentes, que se apresentam na figura abaixo, juntamente com as dependências entre si:

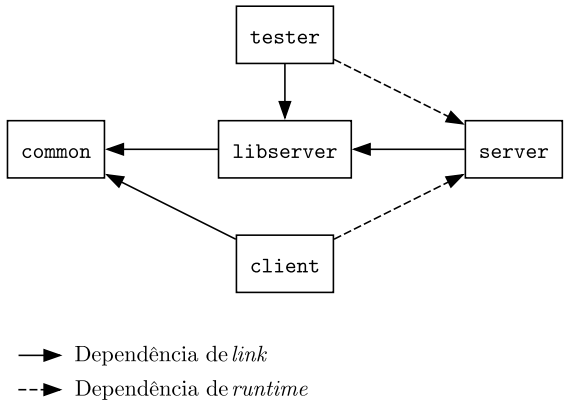


Figura 1: Dependências entre os vários componentes da solução desenvolvida.

A maioria da funcionalidade do *software* está localizada nas bibliotecas `common` e `libserver`:

- `common`
  - Mensagens do protocolo de comunicação entre `client` e `server`, e a sua serialização e deserialização;
  - Lógica de comunicação do cliente.
- `libserver`
  - Várias implementações da estrutura de armazenamento de dados (*backends*), com estratégias de controlo de concorrência distintas.
  - Lógica de comunicação do servidor.

Os programas `client` e `server` não passam de *wrappers* à volta das bibliotecas `common` e `libserver`, respetivamente. No programa `tester`, é implementada uma *framework* para *benchmarking* tanto dos *backends* em `libserver` como da base de dados completa.

2 *Backends* implementados

2.1 `SimpleHashMapBackend`

Este *backend* consiste numa tabela de *hash* que armazena as associações, e cujo acesso concorrente é moderado por um *lock* de leitores e escritores. Há também duas variáveis de condição (associadas ao *lock* de escritores), uma notificada quando os conteúdos da base de dados são alterados, e outra quando todas as operações `getWhen` terminam de verificar se a sua condição é verdadeira.

A operação `getWhen` é a mais complexa: após adquirir o *lock* para escritas, regista-se a *thread* atual

como estando à espera de uma atualização, e espera-se na variável de condição associada a atualizações da base de dados. As operações `put` e `multiPut`, quando terminam, sinalizam as *threads* à espera desta variável de condição (caso as haja). As *threads* na operação `getWhen` podem, caso se verifique a sua condição, remover-se da lista de *threads* à espera e devolver um valor. Quando a última *thread* a executar um `getWhen` verifica a sua condição, sinaliza as *threads* à espera do fim da execução das operações `getWhen` (*threads* no início das operações `put` e `multiPut`).

2.2 `MultiConditionHashMapBackend`

Este *backend* é muito semelhante ao anterior, com a diferença de que são utilizadas mais variáveis de condição relativas à alteração da base de dados, para uma maior granularidade na sinalização de *threads*. As operações `getWhen`, no seu início, criam, caso não exista, uma variável de condição associada à atualização da sua chave, pela qual esperam, e as operações `put` e `multiPut` sinalizam apenas as *threads* à espera de variáveis de condição associadas às chaves modificadas. A execução de uma operação `getWhen` também exige a destruição de uma variável de condição caso não haja mais *threads* dela dependentes.

Uma abordagem alternativa que evita a criação de novas variáveis de condição foi considerada. Esta envolve a criação inicial de N variáveis de condição, às quais as chaves se associam pela sua *hash*. No entanto, testes empíricos demonstraram que esta solução apresenta um desempenho semelhante à abordagem com criação de novas variáveis de condição, não se justificando alterações ao *backend*.

2.3 `ShardedHashMapBackend`

Este *backend* consiste no uso de N tabelas de *hash* (*shards*), associadas a N *locks* de leitores e escritores. A *hash* de uma chave determina o *shard* onde será armazenada. As operações `get` e `put` precisam de adquirir apenas um *lock*, e as operações `multiGet` e `multiPut` precisam de adquirir vários *locks*, fazendo-o ordenadamente para evitar *deadlocks*, e usado *two-phase locking* para diminuir o tempo que cada *lock* se encontra adquirido.

Não foi possível implementar a operação `getWhen` neste *backend* sem prejudicar gravemente o seu desempenho, pelo que esta não foi implementada. Uma operação de escrita precisa de garantir que nada é escrito na base de dados que influencie o resultado das operações `getWhen` a processar, podendo fazê-lo com a aquisição dos *locks* associados aos *shards* das chaves necessárias. No entanto, para conhecer os *locks* que deve adquirir, teria, em primeiro lugar, de adquirir os *locks* associados às chaves a escrever, e adquirir outros *locks* posteriormente poderia causar a violação da ordenação da aquisição de *locks*, conduzindo a uma possível situação de *deadlock*. Este problema poderia ter sido resolvido com um *lock* ao nível da coleção, mas testes empíricos mostraram que o seu uso fazia com que este *backend* tivesse um desempenho semelhante aos *backends* de *lock* único apresentados.

3 Arquiteturas do cliente e do servidor

Na figura abaixo, observa-se uma representação das arquiteturas do cliente e do servidor da base de dados. No cliente, o envio de uma mensagem é feito por qualquer *thread* da aplicação, através de uma escrita no *socket*, cujo acesso é moderado por um *lock*. Há uma *thread* dedicada à exclusivamente à leitura de mensagens do servidor e à sua deserialização. Após um pedido, as *threads* da aplicação bloqueiam numa variável de condição até obterem uma resposta do servidor, que é sinalizada pela *thread* de receção de mensagens. Para diminuir o número de *threads* sinalizadas, são inicialmente

criadas várias variáveis de condição, e uma *thread* escolhe a variável de condição onde bloquear de acordo com o identificador da mensagem enviada.

Quanto ao servidor, uma *thread* em cada conexão é responsável por deserializar os pedidos do cliente e despachá-los para uma *thread pool*, que os executa. A *thread pool* implementada regula o seu número de *threads* para se ajustar ao volume de pedidos dos clientes. As *threads* da *thread pool*, após terminarem as operações na base de dados, respondem aos clientes, escrevendo em *buffers* de mensagens, que são lidos por *threads* de cada conexão, responsáveis apenas por escritas nos *sockets*. Deste modo, é possível que o o resultado de um pedido de um cliente seja difundido por vários clientes, sem que clientes lentos bloqueiem a *thread* na *thread pool*. De momento, nenhum *backend* tira proveito desta funcionalidade.

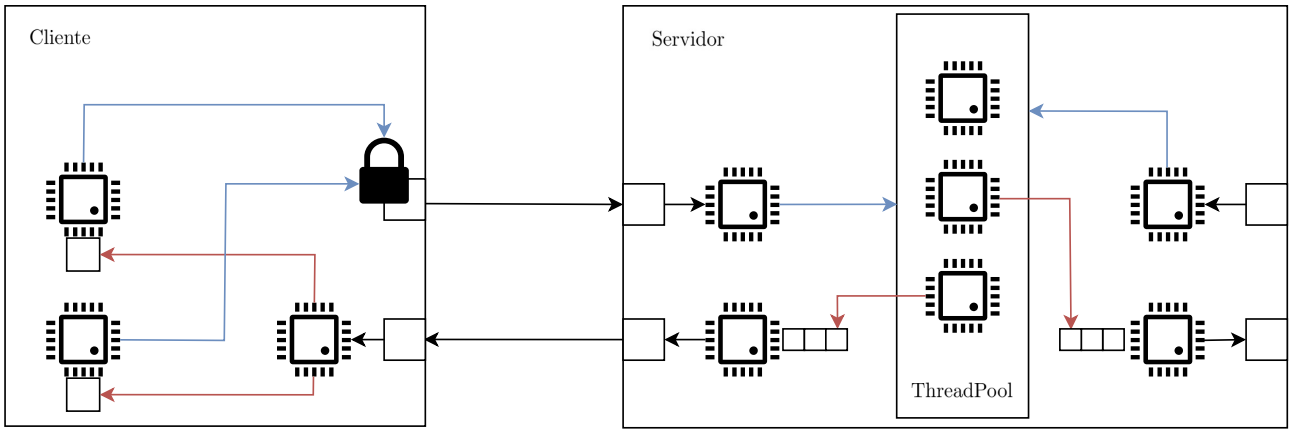


Figura 2: Arquiteturas *multi-thread* do cliente e do servidor.

4 Protocolo de comunicação entre um cliente e o servidor

À direita, apresenta-se um diagrama de sequência de uma possível troca de mensagens entre um cliente e o servidor. Não são representados segmentos da camada de transporte (TCP).

As duas primeiras mensagens permitem a autenticação do cliente: o cliente pede a sua autenticação / registo, e espera que o servidor responda (a autorizar o cliente ou a reportar um erro).

As restantes mensagens correspondem a pedidos do cliente para realizar operações na base de dados, e às respostas do servidor, pelas quais o cliente aguarda. É de notar que todos os pedidos se encontram numerados (campo `id`), e é possível que o servidor envie respostas numa ordem diferente da qual recebeu os pedidos (campo `requestId`), permitindo a existência de clientes *multi-threaded*. Outra nota a fazer é que certas mensagens são utilizadas para vários propósitos. Por exemplo, são utilizadas mensagens do tipo `GetResponseMessage` para responder tanto a pedidos `get` como `getWhen`, uma vez que o conteúdo da resposta é o mesmo.

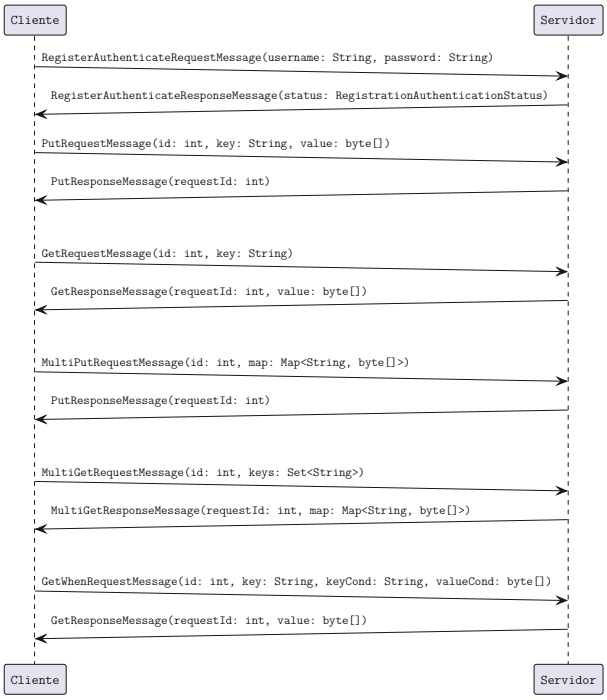


Figura 3: Diagrama de sequência da troca de mensagens entre um cliente e o servidor.

A serialização é feita com recurso às classes `Data[Input|Output]Stream`. Devido à natureza multi-plataforma da linguagem Java, é possível a execução do cliente em diversos sistemas. Ademais, devido à documentação publicamente disponível das classes mencionadas, também é possível interagir com o servidor utilizando outras tecnologias (ex.: C).

5 *Framework* de testes

5.1 Funcionalidade e implementação

Foi desenvolvida uma *framework* para testes de desempenho da base de dados. Um teste de desempenho começa com a criação de uma base de dados e um povoamento inicial. Depois, são criadas várias *threads*, e cada uma gera operações de leitura e escrita aleatoriamente, executando-as e medindo o tempo de execução de cada uma. Estes tempos são utilizados para o cálculo da média e do desvio padrão do tempo de execução de cada tipo de operação. O número total de operações a executar é dividido em blocos, que são dinamicamente distribuídos pelas *threads*. O teste termina quando o número o conjunto de *threads* completa um número predeterminado de operações.

A principal dificuldade na implementação da *framework* relacionou-se com a operação `getWhen`, que bloqueia até uma condição se verificar na base de dados. É possível que todas as *threads* bloqueiem, e não haja nenhuma *thread* disponível para desbloquear as *threads* bloqueadas. Para resolver este problema, é criada uma *thread* no início de cada teste, responsável por desbloquear *threads* bloqueadas em operações `getWhen`. Antes de uma *thread* executar uma operação `getWhen`, atualiza uma variável que armazena a condição na qual irá bloquear. A *thread* de desbloqueio itera constantemente por estas condições, atualizando a base de dados de modo a desbloquear outras *threads*. Assim, não faz sentido que a *framework* de testes meça o tempo médio de execução de operações `getWhen`, visto que este está altamente dependente de quando a *thread* de desbloqueio é escalonada, refletindo fracamente o desempenho da base de dados.

Outra possibilidade de resolução deste problema menos computacionalmente intensiva consiste na existência de uma (ou várias) *threads* responsáveis pela geração das operações, que garantem que todas as operações `getWhen` são correspondidas por uma operação de escrita posterior que desbloqueia a *thread* a executar o `getWhen`. No entanto, esta estratégia pode exigir o uso de blocos muito pequenos para assegurar que uma dada fração das operações sejam do tipo `getWhen`, aumentando o *overhead* do controlo de concorrência, e possivelmente impedindo a *framework* de testes de medir o desempenho máximo da base de dados.

Por último, foram três as bibliotecas utilizadas para auxiliar o desenvolvimento da *framework* de testes. Para implementações de distribuições numéricas (uniforme, Zipf, ...), a biblioteca Apache Commons Math [1] foi utilizada, e as bibliotecas JFreeChart [2] e Apache XML Graphics [3] foram utilizadas para a geração automática dos gráficos apresentados neste documento.

5.2 Metodologia de testagem e resultados

Procurou-se testar o desempenho da base de dados desenvolvida em diversos cenários. Para isso, tirou-se proveito do elevadíssimo grau de configurabilidade da *framework* de testes desenvolvida, e foram concebidos diferentes *benchmarks*, descritos no anexo 8.1, e executados num ambiente de execução descrito no anexo 8.2. Testaram-se os *backends* implementados diretamente, uma vez que a comunicação cliente-servidor apenas adiciona um tempo aproximadamente constante a cada medição, nada revelando e dificultando a comparação entre os *backends*.

Abaixo, observam-se os tempos médios de execução das operações envolvidas no *benchmark* "Maioritariamente leituras". Para qualquer operação e número de *threads*, o desempenho dos *backends* de *lock* único é aproximadamente igual, visto que o *overhead* do controlo de concorrência é o mesmo (aquisição de um *lock* de escrita / leitura conforme a operação). No entanto, o mesmo não é verdade para o `ShardedHashMapBackend`. Com uma *thread*, a necessidade de aquisição de vários *locks* em operações `multiPut` resulta num maior tempo de execução de cada operação. No entanto, com o aumento do número de *threads*, a granularidade dos *locks* reduz a contenção, permitindo que as operações de todos os tipos sejam menos demoradas. Neste *backend*, a partir de 4 *threads*, para operações `multiGet`, o *overhead* de controlo de concorrência é compensado pela menor contenção.

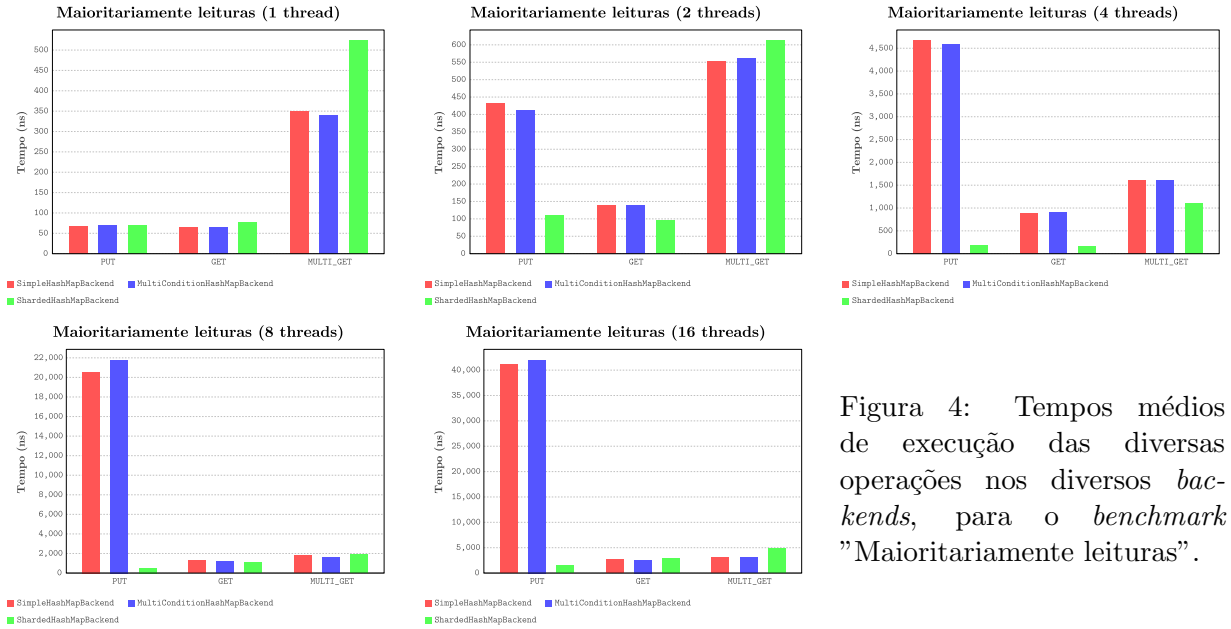
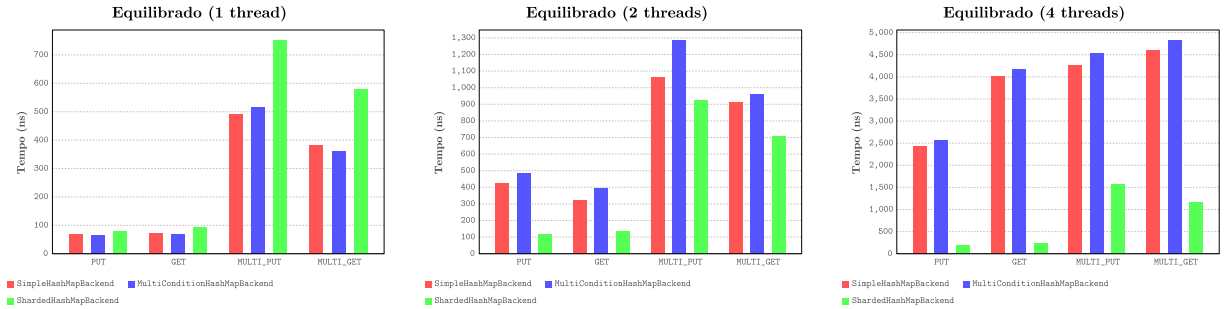


Figura 4: Tempos médios de execução das diversas operações nos diversos *backends*, para o *benchmark* "Maioritariamente leituras".

No *benchmark* "Equilibrado", a situação é semelhante à anterior. O `ShardedHashMapBackend` apresenta um maior *overhead* no controlo de concorrência em operações que envolvem várias chaves, mas que é compensado por uma menor contenção quando uma segunda *thread* é introduzida. A partir de quatro *threads*, nos *backends* de *lock* único, observa-se que as operações `get` são mais demoradas que as `put`. Isto deve-se à injustiça dos *locks* de leitura e escrita utilizados que, em combinação com uma maior competição entre leituras e escritas, dificulta que várias operações de leitura sejam executadas em paralelo. Configurar estes *locks* como justos elimina esta discrepância entre leituras e escritas, mas origina tempos de execução cerca de três vezes maiores para ambas as operações, pelo que se optou por manter os *locks* injustos na implementação dos *backends*.



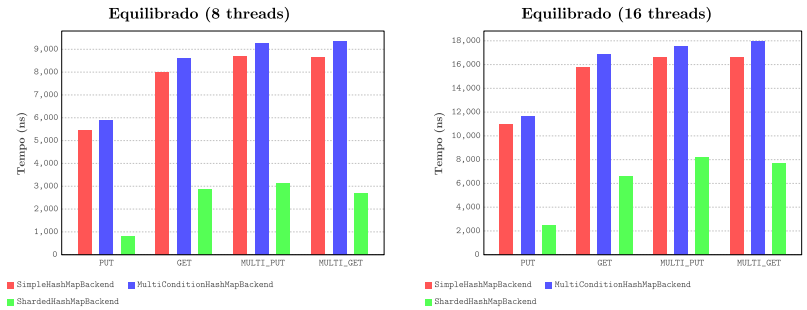


Figura 5: Tempos médios de execução das diversas operações nos diversos *backends*, para o *benchmark* "Equilibrado".

Como mostram os gráficos abaixo, quando são executadas operações `getWhen` em duas *threads* ou menos, os *backends* de *lock* único apresentam sensivelmente o mesmo desempenho, visto que o maior *overhead* do controlo de concorrência do último *backend* nega as suas vantagens. No entanto, quando o número de *threads* aumenta, o uso de variáveis de condição de maior granularidade diminui o tempo de execução das operações de escrita, visto que estas devem aguardar pelo término da execução dos *triggers* (verificações de condições de operações `getWhen`), e estes surgem em menor número. Também se observam melhorias, embora menores, nos tempos das operações de leitura, visto que o menor número de *triggers* a executar faz com que o *lock* para leitura possa ser adquirido, em média, mais rapidamente.

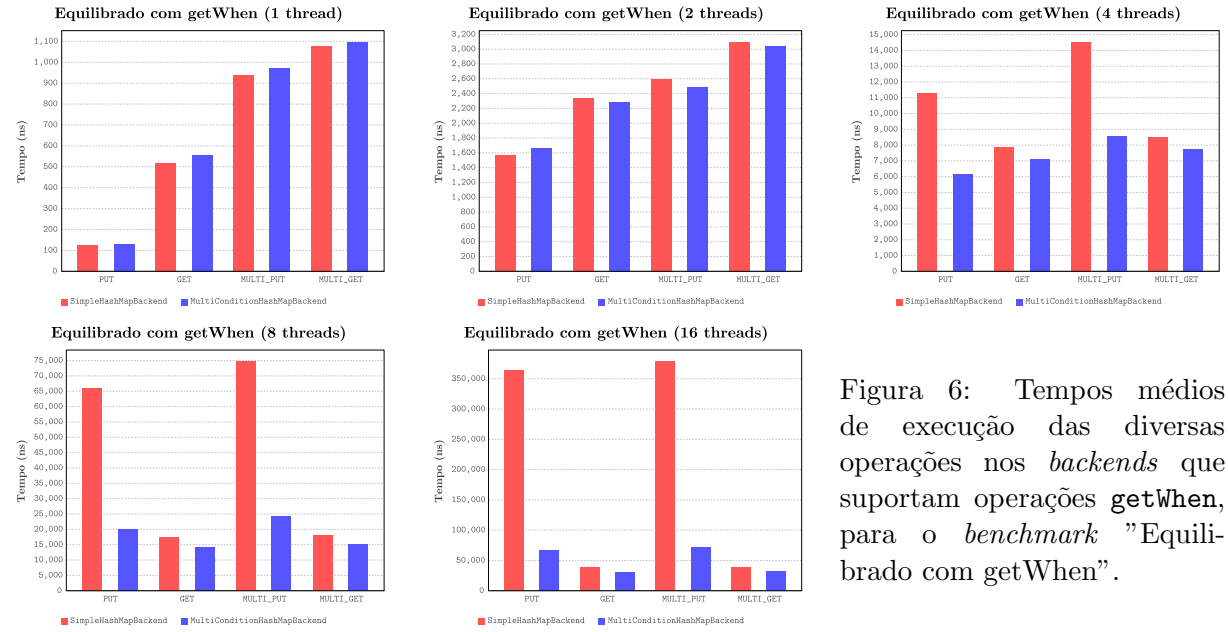


Figura 6: Tempos médios de execução das diversas operações nos *backends* que suportam operações `getWhen`, para o *benchmark* "Equilibrado com getWhen".

Como esperado, e como se observa nos gráficos acima, o tempo médio de cada operação aumenta com o número de *threads*, visto que cada *thread* fica bloqueada em *locks* por mais tempo. No entanto, o que se procura avaliar é o desempenho total de um *backend*, influenciado tanto pelo número de *threads* como pela duração de cada operação. Logo, apresenta-se abaixo a duração de cada teste de acordo com as variáveis que a influenciam. Com a introdução de uma segunda *thread*, nos *backends* de *lock* único, apenas se verificam melhorias de desempenho no *benchmark* "Maioritariamente leituras", visto que estas podem ocorrer em paralelo nestes *backends*, mas o mesmo já não é verdade para escritas. Com o `ShardedHashMapBackend`, como as operações de escrita realizadas não exigem a aquisição de um *lock* sobre a totalidade da base de dados, o melhor desempenho obtém-se para 4 *threads* em ambos os *benchmarks*. A partir de 4 / 8 *threads*, observa-se, de um modo geral, degradação do desempenho, uma vez que é a partir deste ponto que se torna impossível haver afinidade das *threads* de *software* às *threads* físicas do processador utilizado, sendo o desempenho prejudicado pelo tempo de sucessivas trocas de contexto.

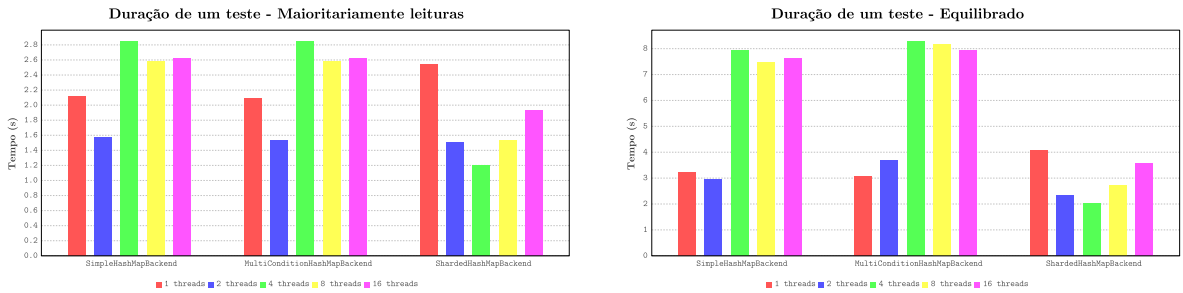


Figura 7: Duração dos testes dependendo do *benchmark*, *backend* e número de *threads* escolhidos.

6 Conclusão

Com base nos os testes de desempenho realizados, conclui-se que a maioria do tempo de execução de cada operação tem origem no controlo de concorrência. O mesmo já poderia não ser verdade caso a base de dados fosse persistente, uma vez que operações em disco são várias ordens de grandeza mais demoradas do que operações em memória. É por este motivo que não foi benéfico, neste contexto, adicionar a operação `getWhen` ao `ShardedHashMapBackend`, mas isso poderia já ser viável caso a base de dados fosse persistente. Estes resultados justificam, por exemplo, o motivo pelo qual o motor de base de dados Redis, também chave-valor e em memória, serializa todos os acessos à base de dados, apesar de permitir I/O concorrente. [4]

Consideramos ter cumprido em pleno os requisitos do enunciado do trabalho prático. No entanto, temos ideias de como melhorar projeto desenvolvido. Além de preocupações com a qualidade de código e documentação, há formas de melhorar o conteúdo do projeto. A título de exemplo, apesar do servidor permitir difusão de mensagens a vários clientes, nenhum *backend* tira proveito desta funcionalidade. Um possível *backend* poderia não bloquear em operações `getWhen`, sendo as respostas a estes pedidos enviadas quando uma escrita é feita na base de dados. Deste modo, não seriam necessárias *threads* do servidor bloqueadas em operações `getWhen`.

7 Bibliografia

[1] The Apache Software Foundation. "Commons Math: The Apache Commons Mathematics Library.", Apache Commons. Accessed: Dec. 20, 2024. [Online.] Available: `https://commons.apache.org/proper/commons-math/`

[2] D. Gilbert. "Welcome To JFreeChart!.", JFreeChart. Accessed: Dec. 20, 2024. [Online.] Available: `https://www.jfree.org/jfreechart/`

[3] The Apache Software Foundation. "Apache FOP." The Apache XML Graphics Project. Accessed: Dec. 20, 2024. [Online.] Available: `https://xmlgraphics.apache.org/fop/`

[4] Alibaba Clouder. "Improving Redis Performance through Multi-Thread Processing.". Alibaba Cloud. Accessed: Dec. 25, 2024. [Online.] Available: `https://www.alibabacloud.com/blog/improving-redis-performance-through-multi-thread-processing_594150`

8 Anexos

8.1 Características do *benchmarks*

	Maioritariamente leituras	Equilibrado	Equilibrado c/ `getWhen`
Número de chaves	1024		
Número de valores	1024		
Comprimento de uma chave	8 caracteres		
Comprimento de um valor	8 octetos		
Percentagem de `put`s	5	25	25
Percentagem de `get`s	70	25	20
Percentagem de `multiPut`s	0	25	25
Percentagem de `multiGet`s	25	25	25
Percentagem de `getWhen`s	0	0	5
Distribuição de chaves	Uniforme		
Distribuição de valores	Uniforme		
Distribuição de número de chaves em operações multi	Uniforme entre 2 e 4, inclusive		
Número de operações	8 milhões		1 milhão[1]
Tamanho de um bloco	4096		

Tabela 5: Características dos *benchmarks* concebidos.

Os três *backends* foram testados, sendo que o `ShardedHashMapBackend` foi inicializado com 64 *shards* e não foi sujeito ao *benchmark* "Equilibrado c/ `getWhen`".

8.2 *Hardware* e *software* utilizados para a execução dos *benchmarks*

Processador	Intel Core i3-7100 3.9 GHz
Memória	8 GiB DDR4 2400 MT/s
Sistema Operativo	Linux 6.11.5
JRE	OpenJDK 21.0.5

Tabela 6: *Hardware* e *software* utilizados para a execução dos *benchmarks*.

[1] Devido à *thread* de desbloqueio de operações `getWhen`, o teste seria muito demorado caso fossem executadas 8 milhões de operações.