Universidade do Minho
Escola de Engenharia

Dezembro 10, 2024

Relatório SD - Grupo 16

Nuno Aguiar (a100480)

Diogo Cunha (a100481)

João Valente (a100540)

Sumário

1) Introdução

No decorrer da cadeira de Sistemas Distribuídos foi-nos solicitada a realização de um projecto cujo objectivo se prende na criação de um serviço de armazenamento de dados partilhado, vulgarmente conhecido por "Cloud". Para o efeito deverá ser implementada uma solução com servidor e clientes, onde estes últimos poderão aceder através de sockets TCP. Funcionalidades mais avançadas por nós implementadas incluem operação de leitura condicional e suporte a clientes multi-threaded. O presente relatório irá explorar a implementação, arquitectura e estratégias de optimização adoptadas pelo grupo.

2) Arquitetura do Programa

Nesta secção, iremos detalhar a arquitetura das componentes principais do sistema — cliente e servidor — com foco nas suas estruturas internas e interações.

2.1) Cliente

Cada cliente contém um *boundedBuffer* do tipo *<CliToSerMSG>* (interface de mensagens que são direcionadas ao servidor e enviadas pelo cliente) de nome *sendbuffer* e uma *linkedlist* que recebe itens do tipo <SavedResponse> (mensagens da interface *ServToCli* encapsuladas com a hora de chegada) de nome *arrivedReply*.

Por fim, cada cliente é composto por **três threads**, sendo que:

- ***RequestHandler***: Processa os pedidos do cliente, encapsulando-os no formato correto e enviando-os para o *boundedBuffer* (*sendbuffer*).
- ***SendMessage***: Lê as mensagens do *boundedBuffer* e envia as mensagens para o servidor através de uma *socket (ClientOutputRunnable)*.
- ***ReadMessage***: Recebe mensagens do servidor pelo *socket (ClientInputRunnable)*, adiciona-lhes a hora de chegada e armazena-as na *linkedlist* ou em formato de ficheiro.

2.2) Server

O server usa a arquitetura ***two-thread-per-connection*** sendo que para cada cliente é disponibilizada uma thread de leitura e uma de escrita, fazendo assim com que a operação de leitura nunca bloqueie a de escrita e vice-versa, permitindo ainda que o server atenda clientes multithreaded. É ainda disponibilizado a cada cliente uma terceira thread que irá estabelecer a comunicação entre o *clientHandler* e o server:

- ***replyThread*** (Envio de Mensagens):
  - Lê as mensagens do *boundedBuffer* (*OutputBuffer*).
  - Serializa e envia as mensagens para o cliente através da *socket(ClientOutputRunnable)*.
- ***readerThread*** (Receção de Mensagens):
  - Recebe mensagens do cliente através da *socket(ClientInputRunnable)*.
  - Adiciona as mensagens recebidas ao *OutputBuffer* após encapsular as mesmas.
- ***handleInputBuffer*** (Encapsulamento e Envio de Pedidos):
  - Lê as mensagens do *OutputBuffer* e finaliza a encapsulação das mesmas adicionando-lhes o seu nível de prioridade (que futuramente servirá para organizar a ordem das tarefas) correspondente ao pedido.
  - Adiciona os pedidos ao buffer global *unscheduledTaks*.

O server contêm um *HashMap* que mantêm o controlo sobre os *buffers* dos clientes de forma a disponibilizar às threads que não estão a comunicar diretamente com eles uma maneira de lhes devolver o resultado dos pedidos realizados por eles.

Além das threads dedicadas exclusivamente a cada cliente, o server contêm também três *threadPools* que irão agendar, realizar e devolver as tarefas realizadas:

- ***`ThreadPool de Schedulers`***: Têm `N`[1] threads a ler de um *`SortedboundedBuffer`*(funciona da mesma maneira que um *`boundedBuffer`* tendo a particularidade que os itens nele armazenados podem ser reorganizados) chamado *`unscheduledTaks`*, assim que chega uma tarefa nova, vai ser agendada por uma das threads que estava a aguardar. Esta irá acordar e procurar o melhor `worker` disponível a quem pode oferecer a tarefa para este a realizar.
- ***`ThreadPool de Workers`***: Existem `N`[1] *threads* que irão realizar os pedidos que lhes são atribuídos, cada worker está a aguardar que seja colocada uma tarefa no seu buffer *`tasks`*, enquanto houverem tarefas no buffer eles irão realiza-las assim que este estiver vazio voltaram a esperar que apareçam novas tarefas. Ao finalizar cada tarefa o worker irá encapsular o resultado com o id do cliente e coloca-o no buffer global *`finishedTasks`*.
- ***`ThreadPool de Dispatchers`***: De forma semelhante às *`threadpools`* anteriores, existem também `N`[1] threads a ler do buffer global *`finishedTasks`*, assim que chega uma tarefa nova será acordada uma das threads em repouso. Esta, por sua vez, procura através da chave do cliente o `boundedbuffer` do mesmo na `HashMap` anteriormente mencionada. Ao enviar a mensagem para o buffer do cliente é utilizado um lock para impossibilitar que vários dispatchers escreveram novas mensagens para o mesmo cliente em simultâneo.

2.2.1) Escalonamento de tarefas pelos workers

As tarefas alocadas aos workers pelos schedulers são organizadas por prioridade, o objetivo é melhorar o desempenho das consultas à base de dados, priorizando as tarefas mais rápidas e onde o tempo passado na secção crítica é menor. Posto isto foram definidos três tipos de prioridades: **high** - para as tarefas mais simples(ex gets, puts e getWhen); **medium** - para as tarefas como os multigets; **low** - para a tarefa que necessita de estar na secção critica durante mais tempo(multiputs). Desta forma cada pedido que chega ao server é encapsulado com o seu nível de prioridade e hora a que chegou. Para garantir que uma tarefa de baixa prioridade não é ultrapassada pelas que têm maior prioridade indefinidamente utilizamos a hora de chegada como desempate sendo que a ordem das tarefas vai ser do maior valor para o mais pequeno, a conta utilizada para a **prioridade** foi ***prioridade inicial + (hora atual - hora chegada)*** garantindo assim que após algum tempo as tarefas de prioridade baixa se vão encontrar com uma prioridade maior que as de prioridade alta. Acrescendo à robustez desta metodologia, foi também adicionado um limite máximo de espera por tarefa, alcançado este, a tarefa passa a ser a próxima na selecção, independentemente de ter um valor de prioridade menor que outras.

2.2.2) Escolha do melhor Worker

De forma a que um dos schedulers escolha o melhor worker disponível sem sobrecarregar um worker ou ter workers inativos muito tempo, decidimos definir um **estado** a cada worker sendo que este pode ser ***Free***, ***Working*** ou ***Capped***. No caso, cada worker irá definir o seu estado através do número de tarefas que têm (ou teve) no seu buffer, desta forma um worker cujo buffer *tasks* esteja vazio terá o seu estado definido como *Free*. A partir do momento em que uma tarefa é colocada no seu buffer o estado muda para working, independentemente da tarefa já estar a ser realizada ou não, sendo que após esta ser finalizada o worker irá consultar o seu buffer e caso este esteja vazio volta a mudar o estado para *Free* caso contrário continuará a realizar as tarefas que lhe foram alocadas. Por fim de forma a que um worker não tenha a sua lista sempre cheia foi definido um limite de tarefas no buffer, este ao ser atingido passa o estado a *capped*, neste caso enquanto o worker não realizar todas as tarefas que lhe foram alocadas o seu estado manter-se-á *capped* até ao momento em que não tiver mais pedidos para fazer e o seu estado será definido como *free* novamente.

[1] `N é definido ao ligar o servidor`

Desta forma os schedulers irão escolher o melhor worker cada vez que estiverem a alocar um pedido, o melhor worker será sempre o que têm o seu estado *free*, em caso de estarem todos com o estado *working* o caso de desempate será o número de tasks que cada um têm no seu buffer, sendo que o melhor worker será sempre o que têm menos. No caso de todos os workers estarem *capped* os schedulers irão aguardar até serem notificados quando pelo menos um dos workers estiver livre.

2.2.3) Autenticação

Visto que neste projeto existe um limite de **N**[1] clientes que se podem encontrar online em simultâneo, tivemos de arranjar uma forma de lidar com isto garantido que é mantida a ordem de chegada ao servidor. A solução encontrada passa por alocar duas threads para este serviço e criar uma lista de espera que funcionará com o método *FIFO*(*first-in-first-out*), desta forma sempre que um cliente se conecta uma das threads coloca-o na lista de espera, a segunda thread verifica se este se pode conectar ou tem de aguardar, caso seja possível o cliente é removido da lista. Assim que existe espaço no servidor é sempre removido o primeiro cliente da lista.

3) Fluxo de Mensagens

De forma a demonstrarmos o fluxo de cada pedido desde o momento em que um cliente (este cliente já tem de se encontrar corretamente autenticado) o faz até ao momento em que este é devolvido decidimos realizar um diagrama:

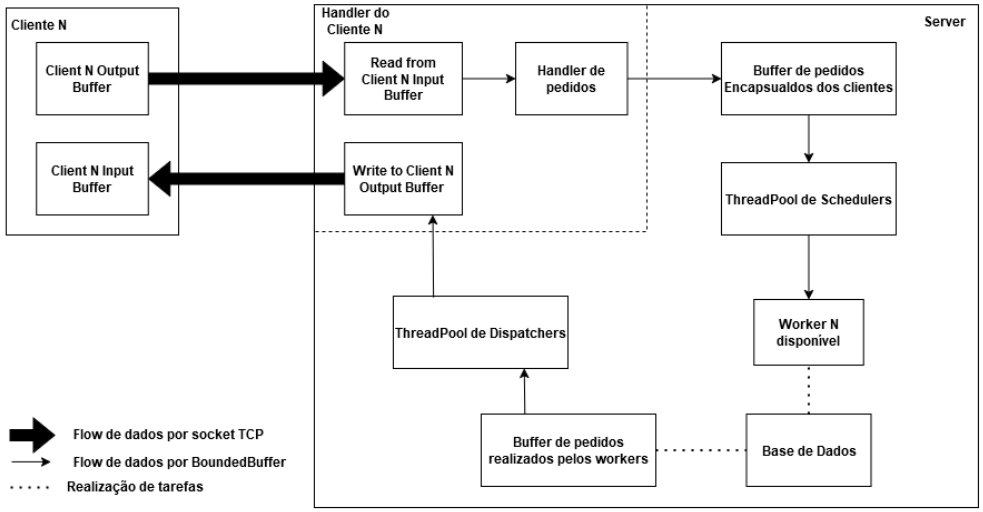

Figura 1: Arquitetura da solução durante o processo de um pedido

Cliente:

1. Escreve tarefa que pretende realizar;
2. Tarefa é encapsulada pelo cliente e coloca-a no seu OutputBuffer;
3. Thread de Output envia a mensagem para o servidor.

Server-ClientHandler

1. Thread de Input respetiva ao cliente recebe a mensagem enviada pelo mesmo colocando-a no buffer do handler de pedidos;
2. Handler de pedidos encapsula a tarefa (adicionado id, hora de chegada e prioridade);
3. Mensagem encapsulada é guardada no buffer de pedidos globais.

Servidor

1. Scheduler acorda e lê a nova tarefa do buffer de pedidos globais;
2. Scheduler procura o melhor worker e aloca-lhe a tarefa;
3. Worker recebe a tarefa e realiza-a;

4. Após worker consultar a base de dados e obter o resultado encapsula-o e coloca-o no buffer de pedidos realizados;
5. Dispatcher acorda e lê a nova tarefa realizada do buffer global;
6. Procura o outputBuffer do cliente a quem a tarefa pertence e adquire o seu lock;
7. Bloqueia o buffer do cliente enquanto lhe está a colocar o pedido finalizado, no fim volta a desbloquear o buffer.

Server-ClientHandler
1. Thread de Output respetiva ao cliente recebe a tarefa finalizada e envia-a para o mesmo.

Cliente:
1. Recebe a tarefa finalizada pela thread que está a ler a socket de input;
2. Desencapsula a resposta e guarda-a na linkedlist do cliente;
3. Quando o cliente escolher a opção de ver os seus pedidos a resposta é colocada no seu UI;

4) Arquitetura da Base de Dados

De forma a conseguirmos testar a performance do sistema decidimos elaborar 3 tipos de bases de dados diferentes para ver como é que a gestão dos *locks* iria influenciar o desempenho em cada uma.

4.1) Base de Dados com um Lock Global

Tal como o nome diz, esta solução é a mais simples e apenas utiliza um *lock* para a base de dados inteira, sendo que cada pedido feito irá bloquear a base de dados completa para o realizar. Esta implementação foi utilizada para verificar quais seriam as implementações base das operações que iriamos utilizar no futuro.

4.1.1) Operações disponíveis

4.1.1.1) Get, Put e MultiPut

Tal como descrito anteriormente, e podemos observar pela Figura 3, estas operações utilizam apenas um *lock*, sendo que após o pedido ser realizado este é libertado

4.1.1.2) MultiGet

De forma a realizar esta operação utilizamos esta implementação simples para testar a forma mais eficiente de realizar um *multiGet* sem bloquear muito tempo a base de dados.

Começamos por realizar o bloqueio da base de dados durante os múltiplos *gets*. Concluímos que não seria a forma mais eficiente de realizar esta operação pois a base de dados ficava bloqueada durante muito tempo impedindo assim as operações mais simples de serem concluídas. Então decidimos bloquear a base de dados para fazer uma cópia da mesma, após a cópia ser realizada, desbloqueamos a base de dados para que outros pedidos possam ter acesso à mesma enquanto procuramos as *keys* pretendidas na cópia que realizamos. Com os testes realizados comprovamos que este método era o mais eficiente entre os dois decidindo então utilizá-lo como base nas próximas implementações da base de dados.

4.1.1.3) GetWhen

De forma a realizar esta operação, tivemos de arranjar uma forma que impedisse a thread que está a realizar o pedido de fazer uma espera ativa mas ao mesmo tempo garantir que assim que chegasse a condição que ela pretende a espera acabe e o pedido seja realizado. Também não queremos que a thread a cada *put* novo acorde e procure na base de dados pela sua condição pois nada nos garante que esta chegará num curto intervalo de tempo.

Para satisfazer estes problemas concluímos que o melhor seria criar uma lista de espera, caso a condição não fosse satisfeita assim que a operação fosse pedida seria colocado na lista a chave e o valor de condição que esta está a aguardar. Por fim atualizamos as operações de *put* e *multiPut* para verificarem se estão a colocar na base de dados alguma condição presente na lista de espera, caso isto se verifique, será enviado um signal para a thread que em espera. Por fim após a thread acordar irá então fazer um get à chave em questão.

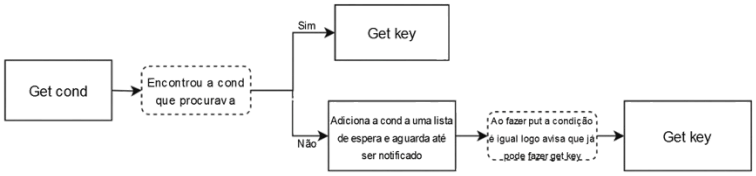

Figura 2: Fluxo da operação *getWhen*

4.2) Base de Dados com uma Batch

De forma a melhorar o desempenho do método anterior decidimos adicionar um segundo *hashmap* que irá servir de *batch*. Esta terá um limite máximo de chaves que consegue suportar ao mesmo tempo, sendo que se este limite for atingido ou ultrapassado irá pausar as operações e fazer um *flush* das chaves para a *hash* principal. O objetivo de termos duas *hashs* é disponibilizar aos workers dois *maps* para consultar sendo que ao existir um lock por *map* podem haver dois workers diferentes a fazer consultas em simultâneo.

4.2.1) Operações disponíveis

4.2.1.1) Get

Esta operação começa por adquirir o *lock* da batch e consultar a mesma, visto que é ai que se encontra a informação mais recente, no caso de encontrar a chave que procura desbloqueia-a visto que já encontrou o que pretendia. Caso contrário ainda com o *lock* da batch adquirido obtém o *lock* da main de forma a garantir a atomicidade dos dados, após este ser adquirido é então desbloqueada a *batch* e faz-se a consulta na main desbloqueando-a assim que esta termina.

4.2.1.2) MultiGet

Esta operação, tal como explicada anteriormente, obtém ambos os *locks* da *batch* e da main hash para fazer uma cópia de cada e por fim fazer a procura das chaves pretendidas após estes *locks* serem desbloqueados.

4.2.1.3) GetWhen

O funcionamento base da operação *getWhen* mantêm-se igual, sendo que a única diferença é o facto de agora não se consultar apenas a base de dados main mas também a *batch*.

4.2.1.4) Put e MultiPut

Assim como no *get*, começam por adquirir o *lock* e fazer put dos dados, caso a quantidade dos dados na *batch* não exceda o limite da mesma, o *lock* é libertado e a operação acaba aí, caso contrário é necessário fazer *flush* dos dados da *batch* para a *hash* principal, sendo então adquirido o *lock* da base de dados principal.

Existem dois tipos diferentes de *flushes* para estas duas operações, um deles acontece caso a *batch* tenha alcançado o limite e apenas insere os dados nela presente na main. Por fim o segundo tipo acontece quando um multiPut insere um número de chaves maior do que a batch suporta, nesse caso começa--se por inserir tudo o que está na *batch* na main, e após estes serem inseridos são então colocadas

todas as chaves do *multiPut* diretamente na main ficando a *batch* vazia e a *hash* principal com os dados atualizados.

4.3) Base de Dados com um Lock Global e Lock por chave

Após testarmos o desempenho das bases de dados anteriores chegamos à conclusão que a diferença entre elas não era tão boa como esperado inicialmente, por isso decidimos implementar um novo modelo na Base de Dados que disponibiliza um lock por chave. Esta base de dados utiliza *DataEntry*, que combina o valor da chave e um *lock* exclusivo, permitindo operações simultâneas em chaves diferentes. Em casos de concorrência, a eficiência é garantida pois as operações de leitura e escrita apenas dependem dos *locks* individuais, sendo o *lock* global apenas utilizado para aceder à estrutura interna.

4.3.1) Operações disponíveis

4.3.1.1) Put e Multiput

Estes métodos acedem à base de dados através de um lock global. Caso os objetos a serem modificados não existam, estes são criados e os respetivos *locks* são obtidos e bloqueados. De seguida o acesso à base de dados volta a ser possível, usufruindo apenas dos *locks* exclusivos para as operações de escrita. A ordem correta de libertação dos *locks* é garantida pois estes são armazenados por ordem de aquisição em um HashMap.

4.3.1.2) Get e Multiget

Estes métodos utilizam o *lock* global para localizar as entradas correspondentes às chaves. Os *locks* exclusivos são novamente armazenados por ordem num HashMap. Após o processo de aquisição dos *locks* e bloqueio dos mesmos, a base de dados é libertada e as operações de leitura são feitas atomicamente com o uso dos *locks* individuais. Novamente a libertação dos *locks* é feita pela mesma ordem de aquisição.

4.3.1.3) GetWhen

Esta implementação mantém-se semelhante, contrastando que a verificação da condição é feita através dos *locks* individuais de cada chave usada na operação, permitindo o acesso concorrente à base de dados.

5) Avaliação de Desempenho

De forma a avaliarmos o desempenho entre as 3 bases de dados desenvolvidas realizamos diversos cenários de conexão entre os clientes e o servidor. Os cenários que decidimos desenvolver testam o desempenho em conexões de 1, 15, 25 e 50 clientes. Como forma a testar a performance do servidor relativamente a diferentes *workflows* utilizados pelos clientes foram etabelecidos 3 *workflows* principais: Multiput-Heavy onde a maioria dos comandos realizados seriam multiput, MultiGet-Heavy sendo que os comandos maioritários seriam os multiget e por fim o *workflow* misto onde recebe um número semelhante de diversos comandos. A escolha dos mesmos justifica-se pois assim percebemos como é que o servidor lida com tarefas mais exigentes caso estas fossem executadas em maioria (MultiPut mais pesada e MultiGet segunda mais pesada).

De seguida podemos verificar a forma como o servidor escala consoante o tipo de base de dados que está a utilizar (durante um *worflow* de multiget). Podemos observar que o servidor que usa um *lock* por chave tem um melhor desempenho visto que os *workers* têm uma espera reduzida pelo *lock* partilhado. A diferença entre o desempenho é mais notável quando o programa é executado pelo WSL.

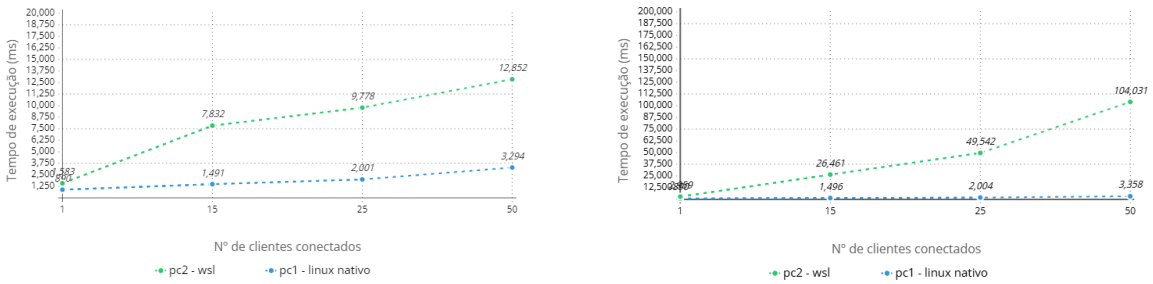

Figura 3: Comparação do desempenho relativamente ao número de clientes ligado ao server

Em relação aos recursos utilizados durante estes processos, podemos verificar que a base de dados que partilha dois *locks* tem um gasto de memória inferior à base de dados que gera um *lock* por chave, uma vez que estes passam a ser diretamente proporcionais ao número de chaves, o que torna o seu tamanho em memória considerável.

Memória por Teste	PC1 - linux nativo - Batch	PC2 - WSL - Batch	PC1 - linux nativo - lock por chave	PC2 - WSL - lock por chave
15 clientes	723 MB	4195 MB	1436 MB	5220 MB
25 clientes	668 MB	4993 MB	1390 MB	5248 MB

Tabela 1: Memória utilizada nos testes de worflow multiget

Nestes resultados não foram utilizados o da Base de Dados com um lock Global visto que esta tem uma performance ligeiramente inferior à da batch na generalidade, sendo que acabamos por disconsiderar os seus resultados. Relativamente a *workflows* diferentes, verificamos que o *workflow* do Multiput demora substancialmente mais tempo que os restantes *workflows*.

De forma a automatizar os testes desenvolvemos diversos scripts na pasta "tests" do projeto, sendo que ao testarnos estes testes numa máquina nova foi sempre necessário dar permissão aos ficheiros .sh. Para consultar os resultados dos testes é criado um ficheiro na pasta logs/metrics-test.

6) Logs

Para conseguirmos visualizar as mudanças que acontecem no servidor, decidimos desenvolver um sistema de logs que cria um ficheiro cada vez que o programa do servidor é executado. Este guarda todos os comandos e as chaves armazenadas com a timestamp da hora em que foi registada. A performance do servidor piorou bastante, sendo que a maior parte do tempo de execução é derivado da escrita de logs no ficheiro. Apesar desta desvantagem achamos necessário ter um método de registar todos os comandos realizados num ficheiro de logs.

7) Conclusão

Este projeto permitiu consolidar o nosso conhecimento sobre os conceitos de serialização e desserialização de mensagens, uso de locks e variáveis de condição, BoundedBuffers e ThreadPools. Aprofundamos conceitos relativos a arquiteturas clientes-servidor e servidor-workers em ambientes multi-threaded, com processamento concorrente de pedidos e respostas.

Ao desenvolver este projeto deparamo-nos com alguns problemas que não conseguimos corrigir dentro do prazo estabelecido e seria o nosso foco caso investissemos novamente neste trabalho. Entre eles foi o funcionamento do comando *getWhen*, visto que quando este comando é feito a comunicar diretamente com a Base de Dados nunca ocorreu nenhum problema, mas a partir do momento em que este é feito pelos workers do servidor, apenas após o servidor receber a condição de que se está à espera uma segunda vez é que a resposta ao *getWhen* é devolvida.