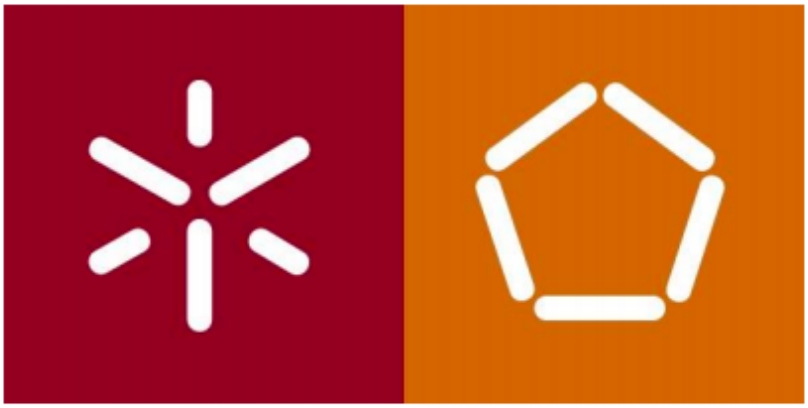

Universidade do Minho

Mestrado Integrado em Engenharia Informática

29/05/2021

Engenharia de Sistemas de Computação
Trabalho Prático 2

António Gonçalves (A85516)
Gonçalo Esteves (A85731)

Contents

1 Introdução

Neste relatório iremos abordar o nosso trabalho e conclusões relativas ao segundo trabalho prático da cadeira de Engenharia de Sistemas de Computação.

Neste foi-nos proposto que criassemos um sistema de bases de dados ligado a um servidor, que recebe pedidos de vários clientes, onde cada cliente poderá efetuar vários pedidos.

Temos ainda como principais requisitos os seguintes dados:

1. O registo deverá ter tamanho fixo de 1024 bytes, e a chave deverá ser um inteiro (64 bits);
2. O volume total dos dados inicial terá de ser 1 TB, tendo em conta a possibilidade de este crescer até mais de 100 TB;
3. Deverá conseguir suportar 10 M de pedidos de acesso para leitura por segundo, bem como 500k pedidos de acesso para escrita por segundo;
4. O servidor deverá responder ao pedido com a latência interna de atendimento, que deve ser inferior a 1 ms em média;
5. O número de clientes na aplicação deverá puder variar entre 1000 e 100.000;

Para além disto, deverá ser efetuada uma análise e pesquisa de forma a encontrar uma solução prática e económica para a aquisição, operação e manutenção do sistema.

2 Implementação do Sistema

Como já foi referido anteriormente, o primeiro objetivo deste trabalho passa por criar um sistema capaz de enviar e receber pedidos de clientes. Assim, iremos agora caracterizar e explicar a nossa implementação, quer dos clientes, quer do servidor em si.

2.1 Cliente

Relativamente ao programa que simula a execução do pedido por parte de um cliente, desenvolvemos, numa primeira fase, algo mais simples, criando um cliente que apenas se liga ao servidor e interage com o mesmo de uma entre duas formas possíveis: ou requisita o valor associado a uma dada chave, ou insere um novo par valor (gerado aleatoriamente) e chave. Em ambos os casos, a chave é passada como argumento. A resposta obtida do servidor é o tempo de execução da função internamente, sendo que este é apresentado no fim. Este programa simples pode ser consultado no ficheiro *client.cpp*. Por forma a executar este programa, e considerando que se correu o comando *make* previamente, basta utilizar o comando ***./bin/client.exe <p ou g> <chave>***, onde ***p*** indica que se deve executar um *put*, e ***g*** indica que se deve executar um get.

Após isto, criamos um novo programa, baseado no anterior, que cria um qualquer número de clientes e faz com que todos eles executem um dado número de pedidos. Tanto o número de clientes a criar, como o número e tipo de pedidos são passados como argumentos iniciais ao programa. Este encontra-se no ficheiro *multiclient.cpp*. De modo a executar o programa bastará correr o comando ***./bin/multiclient.exe <p ou g> <nºclientes> <nºpedidos por cliente>***.

2.2 Servidor

No que se refere à implementação do servidor, esta foi mais complexa, uma vez que este será o responsável por armazenar e fornecer informação sobre os dados. Assim, vamos dividir a sua explicação em duas partes: uma relativa à estrutura de dados desenvolvida para suportar toda a informação, e outra referente à organização do servidor em si, isto é, a forma como recebe e controla os pedidos que recebe.

2.2.1 Estruturação dos Dados

De modo a armazenar toda a informação recebida e a permitir que o acesso à mesma seja feito de forma eficiente, optamos por construir uma *hash table* capaz de armazenar pares chave-valor. Esta tabela de *hash* nada mais é do que um *array* de estruturas criadas por nós e designadas de *Element*. Cada uma destas estruturas possui um *array* de chaves, e um *array* de valores, onde uma chave numa dada posição do *array* de chaves corresponde ao valor na mesma posição do *array* de valores. Para além disto, existem duas outras variáveis que ajudam a controlar o tamanho e utilização dos *arrays*, e um *mutex*, que irá garantir que não há acessos em simultâneo a um mesmo elemento.

```
1 struct Element {
2     long *keys;
3     char **values;
4     int length;
5     int used;
6     std::mutex mut;
7 };
```

Utilizando, tal como já foi referido, um *array* destas estruturas, fomos então capazes de construir uma *hash table*. Esta irá distribuir os pares chave-valor pelos elementos tendo em conta o valor de *hash* da chave, que é simplesmente o resto da sua divisão pelo tamanho da tabela. Assim, não só as inserções vão ser mais rápidas, mas também, e especialmente, a procura de informação vai ser muito mais fácil.

Na inicialização da estrutura garantimos que, numa primeira fase, todos os *arrays* de chaves e valores têm o mesmo tamanho que a *hash table*, dando a sensação de se formar uma espécie de matriz quadrada. No entanto, à medida que as inserções vão ocorrendo, o número de "vagas" disponíveis vai diminuindo, e sempre que, num dado elemento, o número de vagas utilizado é igual ao tamanho dos *arrays*, realoca-se a informação para novos *arrays* com o dobro do tamanho, permitindo a inserção de novos dados. A especificação de tudo isto encontra-se no ficheiro *hashtable.cpp*.

2.2.2 Organização do Servidor

Tendo em conta o servidor criado, podemos dizer que a sua estrutura se divide em quatro partes: a primeira, que trata da ligação do servidor a uma porta, onde irá ficar à escuta de pedidos; a segunda, que é responsável pela inicialização da *hash table* onde irá ser guardada toda a informação (pode ser inicializada apenas com o tamanho inicial ou a partir de um ficheiro de *backup* previamente gerado); a terceira, que trata da geração de duas *threads*, uma responsável por contabilizar quantos pedidos o servidor responde por segundo, e a outra responsável por gerar ficheiros de *backup* a cada 5 minutos (estes ficheiros garantem a persistência parcial dos dados em caso de falha do servidor, e poderão ser utilizados para reinicializar um servidor posteriormente, mas sempre que um é criado, o anterior é apagado); e a quarta, e provavelmente mais importante, que é o ciclo responsável pela aceitação de ligações ao servidor e que cria as *threads* que ficarão responsáveis por responder aos pedidos efetuados nessa ligação. O código do servidor em si pode ser consultado no ficheiro *server.cpp*, enquanto que as diferentes *threads* a utilizar poderão ser consultadas em *serverthread.cpp*. Por forma a executar o servidor, bastará correr um dos seguintes comandos: ***./bin/server.exe s <tamanho da hash table>*** ou ***./bin/server.exe f <nome do ficheiro>***.

3 Análise de Resultados

De modo a analisar o desempenho do servidor por nós criado, realizámos uma série de testes, incidindo mais especificamente no que diz respeito ao número de pedidos atendidos por segundo e a latência interna de atendimento de um pedido. Em qualquer um dos casos, qualquer instância posta à prova foi corrida 5 vezes, sendo que os resultados finais derivaram desses testes. O servidor possuía uma *hash table* com 100 mil elementos, e fora inicializado com 10 milhões de entradas. Os testes foram desenvolvidos num dos nodos 641 do *cluster* SeARCH, da Universidade do Minho, utilizando todos os 16 *cores* disponíveis.

3.1 Pedidos Executados por Segundo

De modo a obter a taxa de pedidos a que o nosso servidor seria capaz de responder por segundo, utilizamos o programa *multiclient*, de modo a executar 10 milhões de pedidos, todos do mesmo tipo, utilizando 10 e 100 clientes. Assim, utilizando 10 clientes, cada cliente iria realizar 1 milhão de pedidos, enquanto que utilizando 100 clientes, cada um realiza 100 mil pedidos. Deve-se realçar que: não se testou para um maior número de pedidos totais, uma vez que o valor utilizado aparentou ser já capaz de maximizar o número de pedidos atendidos por segundo para um dado número de clientes; não se testou para um maior número de clientes, uma vez que utilizando 100 clientes se obtinha os mesmos valores que para quantidades de clientes superiores. Estas conclusões foram obtidas através de alguns "testes-piloto" por nós efetuados, aquando da determinação de quais os melhores valores a utilizar para esta análise.

Os resultados obtidos foram os seguintes.

		Máximo	Média	Mediana
Puts	10 clientes	202705,0	128278,5	127897,3
	100 clientes	722843,0	512649,1	655631,0
Gets	10 clientes	402365,0	222693,2	224069,0
	100 clientes	1131812,0	1081865,3	1123453,0

Table 1: Pedidos atendidos por segundo

Observando a tabela acima, verifica-se rapidamente dois fenómenos:

- primeiro, que o número de *gets* atendidos por segundo é cerca de duas vezes superior ao número de *puts*, tal como seria de esperar, uma vez que os *gets* são efetuados muito mais rapidamente (o que também prova a eficácia da estrutura de dados desenvolvida);
- segundo, que um maior número de clientes a efetuar pedidos em paralelo pode levar a um maior número de pedidos respondidos por segundo, mesmo quando o número total de pedidos efetuados é igual. Tal acontece devido a uma maior incidência de pedidos efetuados por segundo, sendo que enquanto o servidor tiver capacidade de resposta, ele irá responder mais rapidamente, uma vez que não necessita de esperar pelo próximo pedido de um mesmo cliente;

Para além disto, podemos constatar que, utilizando os nodos 641 do *cluster*, já somos capazes de obter valores de atendimento de *puts* por segundo superiores aos que eram requisitados. Isto deve-se em grande parte ao facto de que toda a informação está a ser guardada em memória, o que permite que haja um rápido acesso aos dados e capacidade de modificação dos mesmos, especialmente quando estes estão guardados de forma estruturada.

Apesar disto, pode-se destacar já uma diferença do nosso servidor em relação ao que era pedido, já que a razão entre *gets* e *puts* no nosso servidor é de 2, enquanto que no que era pedido a razão era de 20. No entanto, não podemos afirmar com certezas de que isto é um problema, uma vez que poderão ser os *puts* que estão mais acelerados (algo que pode ser provável, tendo em conta a forma como são efetuados).

3.2 Latência Interna de Atendimento

Posteriormente, procedeu-se à análise da velocidade de execução de um *put* ou *get*, e qual o impacto que o aumento do número de clientes a recorrer ao servidor tem. Para tal, fizeram-se testes com 10 e 100 clientes, e para quantidades totais de pedidos entre os 1000 e os 10000000 (utilizaram-se apenas potências de base 10). Os resultados que se seguem são os valores médios obtidos. O principal motivo para tal advém do facto de que calcular a mediana poderia tornar-se bastante custoso para um grande número de pedidos, o que poderia levar a uma degradação da performance. Assim, e mesmo sabendo que a média poderá não representar com a maior exatidão possível o desempenho do sistema (basta lembrar que, sempre que ocorrer uma realocação de memória obtém-se um pico na latência de um *put*), optamos por utilizá-la, não só de forma a não prejudicar o desempenho do sistema, mas também porque esses picos ocorrem esporadicamente, especialmente quando a *hash table* tem um número elevado de elementos.

		1 000	10 000	100 000	1 000 000	10 000 000
Puts	10 clientes	0,003169	0,00321	0,003625	0,003662	0,00417
	100 clientes	0,00444	0,004809	0,00353	0,004165	0,004609
Gets	10 clientes	0,00146	0,0015742	0,001726	0,001669	0,002075
	100 clientes	0,001876	0,001705	0,001334	0,001164	0,001157

Table 2: Latência interna de atendimento média dos pedidos

Tendo em conta os resultados obtidos, estamos em condições de afimar que, na grande maior parte dos casos, os pedidos serão executados internamente em menos de 1ms. No entanto, é sempre bom relembrar que irão ocorrer picos, devido, por exemplo, às realocações de memória ou em casos em que haja mais do que um cliente a tentar aceder ao mesmo elemento da *hash table*.

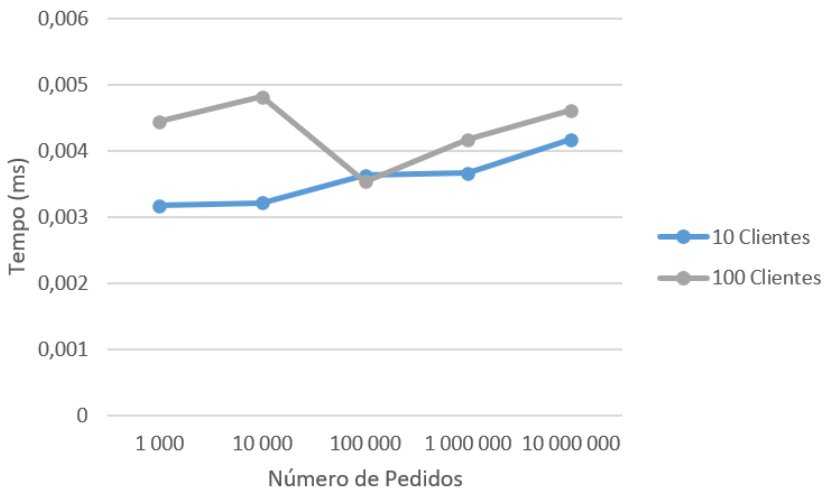

Figure 1: Latência interna de atendimento média de um *put*

Considerando os resultados obtidos com os *puts*, verifica-se que, de um modo geral, os tempos de execução são um pouco superiores para 100 clientes. Isto poderá dever-se a dois fatores: ou dois clientes estão a tentar aceder ao mesmo elemento, então um deles terá de ficar à espera, ou então o uso de muitos *mutex* em simultâneo, que poderá atrasar um pouco a execução do programa.

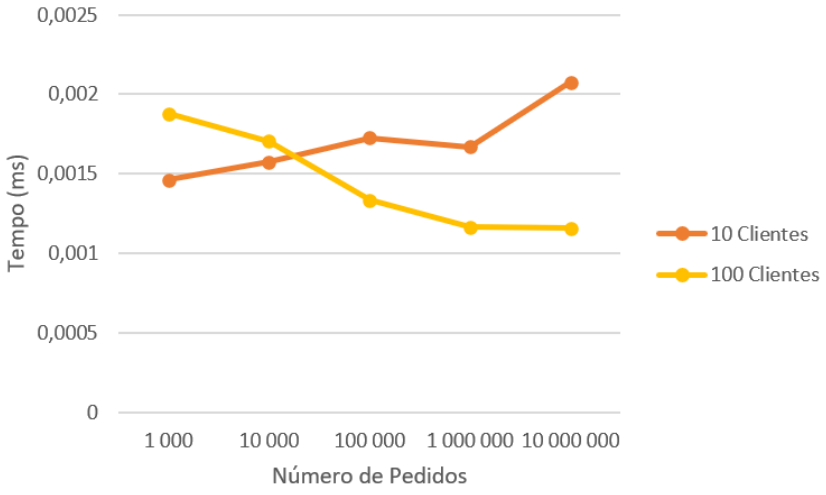

Figure 2: Latência interna de atendimento média de um *get*

No que toca aos *gets*, é observável que o aumento do número de pedidos leva a uma diminuição da latência para 100 clientes, e um aumento da latência para 10 clientes.

4 Análise do Custo do Sistema

Por forma a tentar idealizar um possível sistema capaz de suportar a nossa aplicação, devemos ter em conta não só os resultados obtidos nos testes, mas também as especificações da máquina utilizada para testar. Assim sendo, algumas das mais interessantes poderão ser o número de *cores* (16 físicos, 32 virtuais), a frequência (2.6GHz de base, 3.4GHz máximo) e a memória do processador (64 GB).

Tendo isto em conta, uma possível abordagem ao problema poderá ser aumentar o número de *cores* de modo a que o número de *gets* atendidos por segundo atinja os valores esperados (não é necessário olhar para os *puts* uma vez que estes já se encontram nos valores pretendidos, no entanto, é importante frisar que estes também sairão beneficiados com o aumento do número de *cores*). Ora bem, uma vez que o valor pretendido é de 10M/s, e o valor obtido com a mediana foi de 1123453/s, podemos afirmar que a razão entre os dois é de aproximadamente 9. Considerando que o programa correu em 16 *cores* físicos (iremo-nos para já abstrair do facto de que os clientes e os servidores estavam a correr na mesma máquina, de forma a não tornar as estimativas demasiado baixas), devemos afirmar que iremos necessitar de uma máquina com aproximadamente 144 *cores* físicos, caso as especificações da mesma sejam semelhantes à do nodo do *cluster* utilizado. Para além disto, a máquina irá precisar de cerca de 100TB de memória RAM, uma vez que a informação fica guardada em memória, e 200TB de memória em disco (de modo a guardar duas cópias de segurança em simultâneo, já que a mais antiga só é apagada depois da mais recente ter sido criada). Analisando as *shapes* disponibilizadas pela *Oracle*[2], bem como os custos a elas associadas[1], pensamos ter chegado a uma possível solução.

Utilizando uma *bare metal shape* do tipo **BM.Standard.E4** conseguimos um sistema com 128 OCPU's (cada OCPU corresponde a um *core* físico num processador *Intel* que permita *multi-threading*). Se considerarmos agora o facto de que a frequência do processador *AMD EPYC 7J13* desta *shape* é superior à do processador *Intel E5-2650v2* do nodo 641 do *cluster*, e também que os clientes não irão estar a correr nesta máquina, pensamos que é seguro afirmar que esta *shape* nos irá permitir obter os resultados que pretende-

mos. No entanto, para além disto, teremos de considerar também o custo associado à aquisição de 100TB de memória, bem como o custo da aquisição de *block volumes* suficientes para perfazer um total de 200TB de memória em disco, distribuída por todos os OCPU's (não conseguimos encontrar informações sobre o custo destes últimos, no entanto).

Assim sendo, o custo por hora do nosso sistema é dado pela seguinte expressão:

$$Custo = 1shape \times 128OCPU's \times 0,0224525euro/h + 100TB \times 0,00134715euro/GB = 140,82euro/h$$

Infelizmente, o custo final está mais elevado do que o que gostaríamos, principalmente devido ao facto de que será necessário recorrer a muita memória, o que aumenta os custos finais da solução. Uma implementação que fosse capaz de armazenar informação em disco (não apenas cópias de segurança, mas sim a informação à medida que ia sendo inserida) teria certamente muitos menos custos ao nível da memória. No entanto, o número de instruções por segundo seria inferior ao demonstrado pela nossa aplicação, para uma mesma máquina, uma vez que teria de aceder a informação guardada em ficheiro. Assim sendo, também iria necessitar de uma máquina mais avançada, ou com mais *cores* do que aquela que foi por nós apresentada, sendo que o custo final iria ser maior nesse parâmetro.

5 Conclusão

Com o desenvolvimento deste trabalho fomos capazes de desenvolver as nossas aptidões no desenvolvimento de aplicações paralelas e na análise do desempenho das mesmas. Para além disto, fomos confrontados com uma pequena simulação de uma situação real, em que foi necessário compreender quais as vantagens e desvantagens de recorrer a uma dada técnica ou até mesmo peça de *hardware*, em detrimento de outra.

No entanto, sentimos que não conseguimos explorar o trabalho na sua máxima extensão, devido a algumas dúvidas que foram aparecendo especialmente no que toca à transição da aplicação para um sistema com maior capacidade computacional e mais adequado ao seu uso.

References

[1] Oracle. *Compute Pricing*. URL: https://www.oracle.com/pt/cloud/compute/pricing.html?fbclid=IwAR1r9fumGm2JClqrOcRQhJZDrul9i_afuI2zPeNukIEFerygMtBMWSfzqgk. (accessed: 28.05.2021).

[2] Oracle. *Compute Shapes*. URL: https://docs.oracle.com/en-us/iaas/Content/Compute/References/computeshapes.htm#Compute_Shapes. (accessed: 28.05.2021).