

Relatório TP | Sistemas Distribuídos

Grupo 15 | 2024/2025

João Lobo
(A104356)

Mariana Rocha
(A90817)

Mário Rodrigues
(A100109)

Rita Camacho
(A104439)

Índice

1. Introdução	3
2. Arquitetura do serviço	3
3. Funcionalidades	4
3.1. Autenticação e registo do utilizador	4
3.2. Operações de escrita e leitura simples	4
3.3. Operações de escrita e leitura compostas	4
3.4. Operações de leitura condicional	4
3.5. Limite de utilizadores concorrentes	4
3.6. Multi-threaded	4
4. Programa Servidor	4
5. Biblioteca do cliente	5
6. Interface do utilizador	5
7. Avaliação de desempenho	6
7.1. Get 1000 pares chave-valor aleatórios	6
7.1.1. 10 <i>shards</i> e 1 cliente	6
7.1.2. 10 <i>shards</i> e 3 clientes	7
7.1.3. 100 <i>shards</i> e 10 clientes	7
7.1.4. 1000 <i>shards</i> e 10 clientes	8
7.1.5. 1000 <i>shards</i> e 100 clientes	8
7.2. Get 1000 operações na mesma chave	9
7.2.1. 10 <i>shards</i> e 1 cliente	9
7.2.2. 10 <i>shards</i> e 3 clientes	9
7.2.3. 100 <i>shards</i> e 10 clientes	10
7.2.4. 1000 <i>shards</i> e 10 clientes	10
7.2.5. 1000 <i>shards</i> e 100 clientes	11
7.3. MultiGet 1000 pares chave-valor aleatórios	11
7.3.1. 10 <i>shards</i> e 1 cliente	11
7.3.2. 10 <i>shards</i> e 3 clientes	12
7.3.3. 100 <i>shards</i> e 10 clientes	12
7.3.4. 1000 <i>shards</i> e 10 clientes	13
7.3.5. 1000 <i>shards</i> e 100 clientes	13
7.4. MultiGet 100 operações na mesma chave (10 chaves por multiget)	14
7.4.1. 10 <i>shards</i> e 1 cliente	14
7.4.2. 10 <i>shards</i> e 3 clientes	14
7.4.3. 100 <i>shards</i> e 10 clientes	15
7.4.4. 1000 <i>shards</i> e 10 clientes	15
7.4.5. 1000 <i>shards</i> e 100 clientes	16
8. Conclusão	17

1. Introdução

Este projeto tem como intuito implementar um serviço de armazenamento de dados partilhado, baseado no modelo chave-valor, onde a comunicação entre o servidor e os clientes é realizada através de *sockets* TCP. O servidor encontra-se preparado para lidar com várias conexões ao mesmo tempo de forma eficiente, garantindo acessos concorrentes.

O objetivo principal deste projeto centra-se na criação de um sistema robusto e eficiente, que minimize contenções e otimize o número de threads, de forma que privilegiámos soluções que reduzem a sobrecarga do servidor em cenários de alta concorrência.

De seguida, apresentámos a arquitetura desenvolvida, as funcionalidades propostas, bem como uma análise de desempenho.

Este relatório contém mais páginas do que o inicialmente solicitado, devido à inclusão de muitas imagens detalhadas das avaliações de desempenho.

2. Arquitetura do serviço

A arquitetura do sistema desenvolvido adota uma abordagem cliente-servidor, sendo esta uma arquitetura centralizada, onde o servidor é o núcleo do armazenamento de dados e os clientes conectam-se para realizar operações de leitura e escrita. O servidor é implementado para ser acedido de forma concorrente, tendo em atenção a múltiplos clientes simultaneamente através de threads dedicadas para tal. A comunicação entre o servidor e o cliente é através de um *socket* TCP **por conexão**, permitindo que os clientes realizem diversas operações exploradas de seguida, este protocolo é realizado em formato binário, onde recorremos apenas a **Data[Input|Output]Stream**. A escolha deste protocolo de comunicação foi realizada tendo em conta que o TCP garante uma comunicação estável e confiável entre o cliente e o servidor.

Uma das principais características da arquitetura é o uso de *buckets* (ou *shards*) para armazenar a informação, invés de usar um único *Map* global. São usados dois tipos de *buckets*: um para armazenar as informações de autenticação dos clientes e outro para guardar os dados (chaves e valores), o que permite uma melhor organização da informação e desempenho.

Cada *bucket* contém um subconjunto de pares chave-valor e é acedido de forma independente, melhorando assim a concorrência do sistema. Cada chave de entrada é processada por uma função *hash*, que determina a qual *bucket* pertence, esta função garante uma distribuição equilibrada, reduzindo a possibilidade de sobrecarga em *shards* específicos.

Esta divisão permite que muitas operações de leitura e escrita sejam realizadas de forma paralela, o que aumenta a eficiência e evita a contenção.

A utilização de *buckets* oferece uma melhor distribuição de dados e gestão de memória, tal como reduz a contenção nas operações de escrita, otimizando também o desempenho de leitura.

Do lado do cliente, foi desenvolvida uma biblioteca que encapsula a comunicação com o servidor e facilita a execução das operações.

Para complementar a funcionalidade do projeto, foi também desenvolvido um programa de testes e monitorização do desempenho, demonstrado com o auxílio da ferramenta *benchmarking* que analisa o comportamento do sistema em diferentes *workloads*, sendo posteriormente gerados gráficos que ajudam a visualizar o impacto de diversos fatores.

3. Funcionalidades

O sistema oferece um conjunto de funcionalidades focadas no armazenamento de manipulação de dados de forma eficiente e segura, dando especial atenção à concorrência e ao desempenho. Todas as operações realizadas no servidor são **atómicas**, o que garante a integridade dos dados.

3.1. Autenticação e registo do utilizador

Uma das principais funcionalidades do sistema é a autenticação e registo de utilizadores que garante um acesso seguro ao serviço de armazenamento. Ao autenticar um utilizador, o servidor consulta um *bucket* específico para validar o seu *log-in*, onde apenas utilizadores registados têm acesso às restantes funcionalidades.

3.2. Operações de escrita e leitura simples

As operações de leitura e escrita simples envolvem interações diretas com pares chave-valor no sistema. A operação de **escrita** é realizada através da função `put`, esta armazena um valor associado a uma chave na respetiva *shard*. A operação de **leitura** é realizada com a função `get`, que retorna o valor associado a uma chave específica. Ambas as operações são atómicas, o que garante que o acesso dos dados é seguro mesmo num ambiente com múltiplas *threads*.

3.3. Operações de escrita e leitura compostas

Além das operações simples, o sistema fornece **operações compostas**. A função `multiGet` recebe como argumento várias chaves e retorna os valores correspondentes às chaves fornecidas num mapa. A função `multiPut` realiza a escrita de vários pares chave-valor de forma atómica. O uso destas operações compostas proporciona uma forma eficiente de inserir e ler vários dados de uma vez.

3.4. Operações de leitura condicional

O sistema também possibilita operações de **leitura condicional** através da função `getWhen`. Isto permite que o cliente consiga obter o valor de uma chave específica, quando uma condição associada a outra chave for realizada. A thread aguarda até que o valor da chave condicional corresponda ao valor pedido.

3.5. Limite de utilizadores concorrentes

Também temos implementado no servidor um limite máximo de número de conexões ativas que podem ser mantidas em simultâneo. Esta funcionalidade assegura que o sistema não seja sobrecarregado, dependendo do valor escolhido. Quando o limite de utilizadores é atingido, as novas conexões ficam em espera e aguardam que algum dos clientes ativos encerre a conexão com o servidor.

3.6. Multi-threaded

Por fim, o servidor suporta múltiplas threads do cliente. Com o auxílio da `TaggedConnection` e do `Demultiplexer`, o cliente pode enviar diversos pedidos de forma independente, mesmo quando um dos seus pedidos está bloqueado no servidor. O servidor processa os pedidos de forma paralela, evitando *deadlocks*.

4. Programa Servidor

O programa implementa um servidor *multithreaded* que gere as conexões dos clientes utilizando um sistema de *sharding* para distribuir eficientemente os dados entre partições. A estrutura principal é composta pela classe `Server`, responsável por aceitar conexões até um limite confi-

gurado, fornecendo a cada cliente uma instância de `ServerWorker`, que processa as requisições recebidas. O servidor utiliza uma base de dados segmentada em *shards*, tanto para dados quanto para os utilizadores, o que permite acesso concorrente eficiente e sincronizado. Cada *shard* possui mecanismos de controlo de concorrência, como bloqueios de leitura e escrita, garantindo a consistência e segurança das operações realizadas por múltiplas threads simultaneamente.

As operações suportadas incluem autenticação, registo, inserção e recuperação de dados, bem como consultas condicionais. A lógica de sincronização utiliza enumerações como `RequestType` para classificar os diferentes tipos de requisição, e as operações são coordenadas com bloqueios e condições. Um destaque é a funcionalidade de operações condicionais (`GetWhenRequest`), que permite aos clientes aguardar até que uma condição seja satisfeita. Este recurso utiliza *threads* separadas para controlar as alterações nos valores e sinalizar quando as condições são cumpridas, garantindo eficiência na execução.

O programa também controla rigorosamente o número de conexões ativas, utilizando bloqueios para gerir *slots* disponíveis e notifica as *threads* que aguardam pela libertação da conexão. O *design* do servidor é robusto, priorizando a segurança no acesso a dados, a flexibilidade para atender a diferentes tipos de requisições e a eficiência no processamento de operações simultâneas, tornando-o adequado para sistemas distribuídos com alta concorrência.

5. Biblioteca do cliente

A `ClientLibrary` foi desenvolvida para facilitar a comunicação entre o cliente e o servidor, utilizando uma arquitetura baseada em requisições assíncronas e mensagens com *tags* únicas. Esta gera a conexão de rede através da `TaggedConnection`, que permite enviar e receber dados de forma estruturada. A comunicação é sincronizada com o uso de bloqueios e de condições, garantindo que o cliente aguarde corretamente pelas respostas do servidor sem interferir nas operações simultâneas. A biblioteca oferece funcionalidades como autenticação de utilizadores, registo, inserção de dados e operações de recuperação, incluindo suporte a múltiplas chaves.

Além das operações básicas de chave-valor, a `ClientLibrary` também oferece suporte a operações condicionais, como o método `getWhen`, que permite que o cliente espere até que uma condição específica seja satisfeita no servidor. Isso é útil em cenários que é necessário sincronizar diferentes clientes ou o cliente com o servidor. A biblioteca foi projetada para ser eficiente e escalável, utilizando múltiplas *threads* para gerir a leitura e a escrita de dados de forma não bloqueante. O cliente pode desconectar-se de forma limpa, enviando uma mensagem de desconexão para o servidor. Em resumo, a `ClientLibrary` oferece uma *interface* robusta e eficiente para interagir com um servidor de forma simples e segura.

6. Interface do utilizador

A *interface do utilizador* foi desenvolvida para permitir a interação com o programa criado, sendo apresentadas várias opções dentro das funcionalidades presentes no mesmo, permitindo assim a indicação da(s) operação(ões) para as executar.

Esta oferece uma interação intuitiva permitindo aos utilizadores registar, autenticar e executar operações sobre dados. O programa inicia com a conexão ao servidor e apresenta um menu inicial com opções para registo, autenticação ou saída. Após o *login* bem-sucedido, os utilizadores têm acesso a uma série de funcionalidades, como `put`, `get`, `multiPut`, `multiGet` e `getWhen`. Estas operações permitem manipular pares chave-valor, realizar inserções e consultas condicionais, tudo em tempo real, através de um sistema que utiliza *threads* para executar operações de forma assíncrona, otimizando a experiência do utilizador.

A *interface* também é robusta na gestão de entrada do utilizador, verificando continuamente valores inválidos para evitar erros. Além disso, a organização do código facilita a expansão futura com novos recursos. O uso de *threads* para autenticação e execução de operações complexas, como `getWhen`, destaca um design focado em eficiência e paralelismo, enquanto a inclusão de mensagens detalhadas para erros e resultados de operações aprimora a transparência e o controlo do utilizador sobre as interações realizadas. Ao final de cada sessão, os recursos são libertados de forma cuidadosa, as conexões são encerradas, bem como as *threads*, garantindo a estabilidade do sistema.

7. Avaliação de desempenho

Os testes de desempenho avaliam o comportamento do sistema para diferentes valores de carga, de modo que variámos o número de *shards* e clientes em simultâneo. Foram analisados 4 cenários: o primeiro consiste em realizar o get de 1000 pares chave-valor aleatórios, o que avalia o desempenho do sistema em operações de leitura. No segundo cenário, temos o get de 1000 operações na mesma chave, conseguindo assim simular um *hotspot* e como é que o sistema lida com contenção e bloqueios frequentes. O terceiro cenário é o MultiGet de 1000 pares chave-valor aleatórios, este mede a eficiência das operações que acedem a múltiplas chaves. O último cenário - MultiGet de 100 operações na mesma chave - combina o comportamento de *hotspot* com operações de múltiplas leituras.

Cada tipo de cenário foi testado com 5 configurações distintas: 10 *shards* e 1 cliente; 10 *shards* e 3 clientes; 1000 *shards* e 10 clientes e 1000 *shards* e 100 clientes. Estes testes foram realizados em cada máquina de cada elemento do grupo, com medições detalhadas do tempo de resposta. As especificações de *hardware* de cada máquina estão apresentadas de seguida para contextualizar os resultados obtidos.

Máquina	CPU	RAM	Sistema Operativo	Armazenamento
1	Intel Core i7-13700H	16 GB DDR5	Ubuntu 22.04.5 LTS	1 TB SSD
2	Intel Core i7-1165G7	8 GB DDR4	Arch Linux	512 GB SSD
3	Intel Core i7-1260P	16 GB DDR5	Ubuntu 22.04.1 LTS	1 TB SSD
4	Intel Core i7-1260P	16 GB DDR5	Ubuntu 22.04.1 LTS	1 TB SSD

7.1. Get 1000 pares chave-valor aleatórios

Este *workload* avalia o desempenho ao aceder a chaves distribuídas aleatoriamente, simulando um cenário de leitura generalizada.

7.1.1. 10 shards e 1 cliente

O desempenho é consistente e apresenta tempos de resposta baixos devido à baixa concorrência e pequeno número de *buckets*, minimizando a sobrecarga de busca.

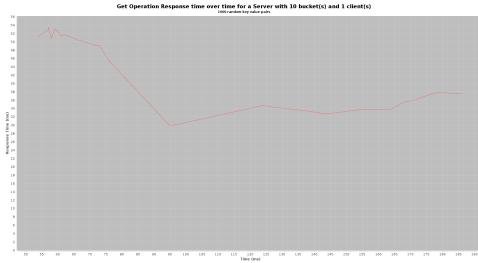


Figura 1: Máquina 1

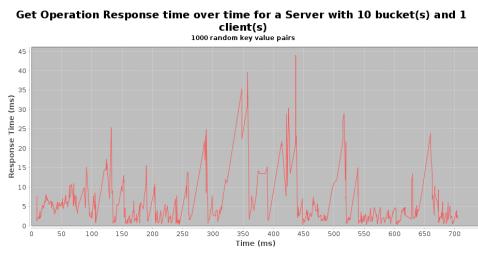


Figura 2: Máquina 2

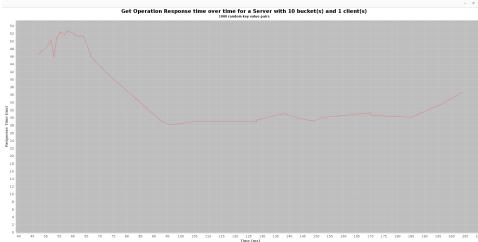


Figura 3: Máquina 3

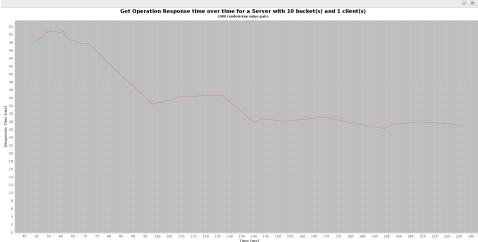


Figura 4: Máquina 4

7.1.2. 10 shards e 3 clientes

Observam-se tempos de resposta mais baixos, no entanto há uma maior variância nos valores.

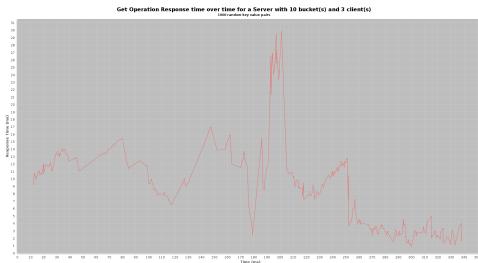


Figura 5: Máquina 1

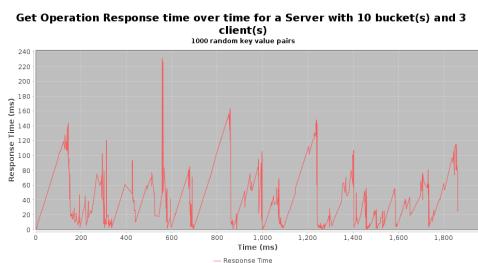


Figura 6: Máquina 2

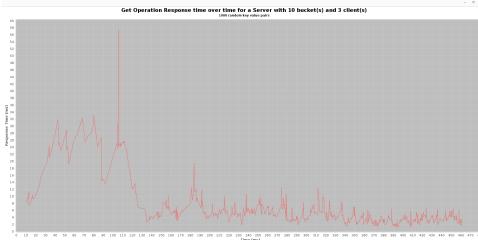


Figura 7: Máquina 3

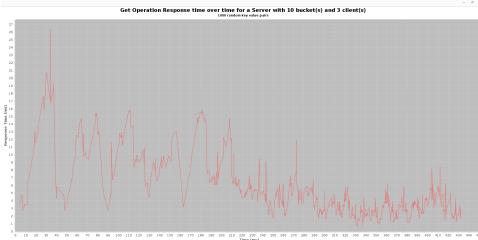


Figura 8: Máquina 4

7.1.3. 100 shards e 10 clientes

O aumento de buckets começa a gerar *overhead* na localização das chaves, impactando levemente o desempenho, especialmente em picos de carga.

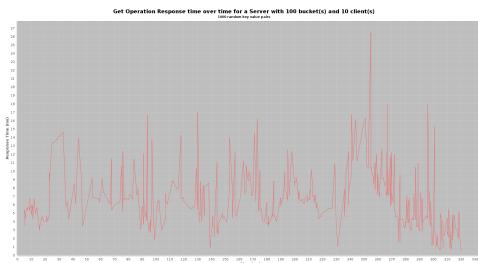


Figura 9: Máquina 1

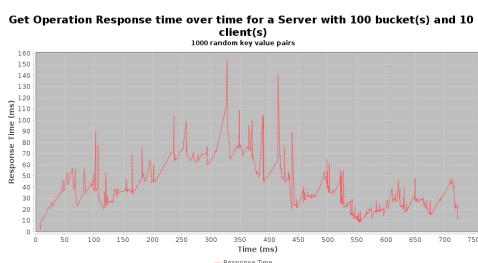


Figura 10: Máquina 2

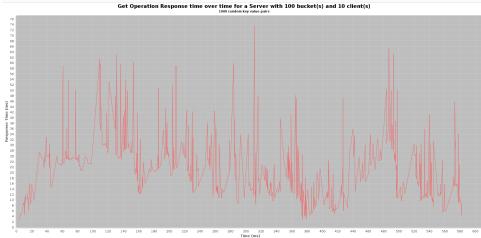


Figura 11: Máquina 3

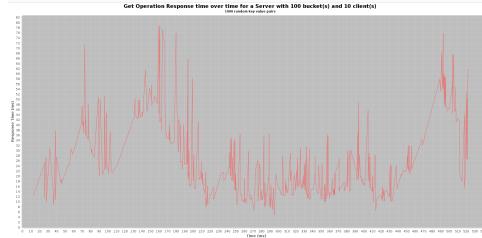


Figura 12: Máquina 4

7.1.4. 1000 shards e 10 clientes

A sobrecarga ao gerir tantos *buckets* afeta significativamente os tempos de resposta, apesar do número moderado de clientes.

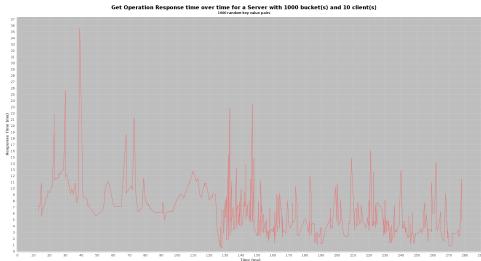


Figura 13: Máquina 1

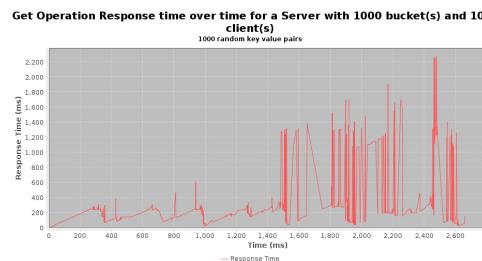


Figura 14: Máquina 2

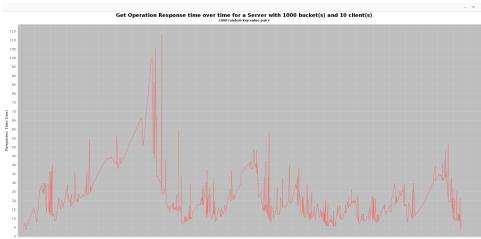


Figura 15: Máquina 3

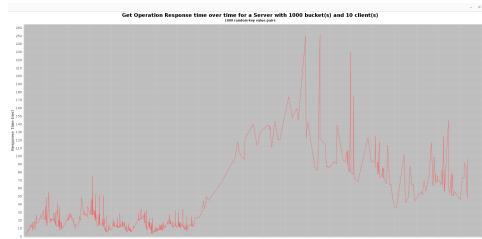


Figura 16: Máquina 4

7.1.5. 1000 shards e 100 clientes

O sistema enfrenta severos problemas de desempenho devido à combinação de alta concorrência e complexidade na gestão de muitos *buckets*.

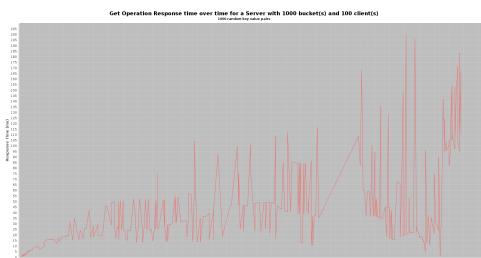


Figura 17: Máquina 1

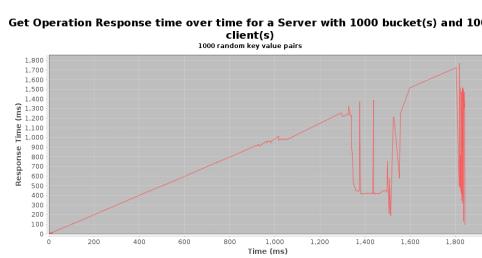


Figura 18: Máquina 2

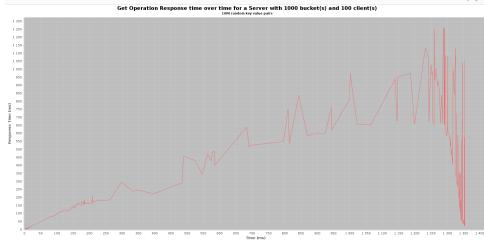


Figura 19: Máquina 3

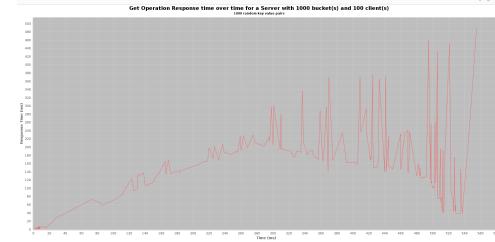


Figura 20: Máquina 4

7.2. Get 1000 operações na mesma chave

Este *workload* mede o comportamento do sistema em cenários de acesso concentrado, que podem gerar *hotspots*.

7.2.1. 10 shards e 1 cliente

O tempo de resposta é excelente, já que se trata de um processo sequencial visto se tratar de apenas um cliente à procura do mesmo valor, nem há sobrecarga significativa de gestão.

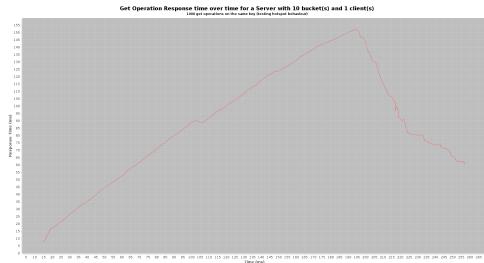


Figura 21: Máquina 1

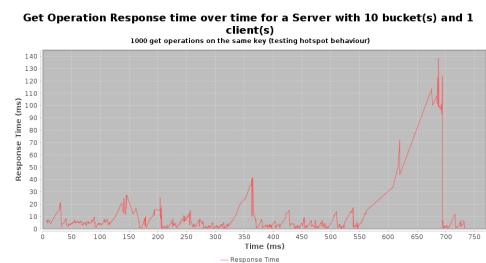


Figura 22: Máquina 2

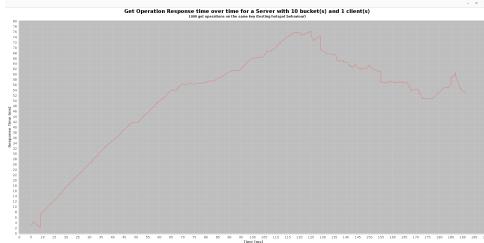


Figura 23: Máquina 3

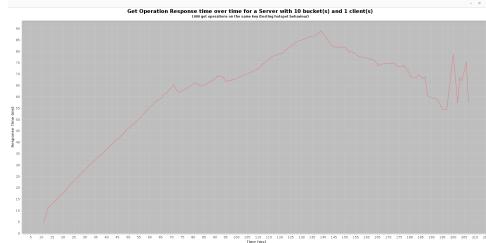


Figura 24: Máquina 4

7.2.2. 10 shards e 3 clientes

Um leve aumento na latência é percebido, mas o sistema mantém boa eficiência na execução de operações repetitivas. Podemos observar que ao longo do tempo o valor do tempo de resposta tem tendência a diminuir, visto poder dar uso da concorrência entre clientes.

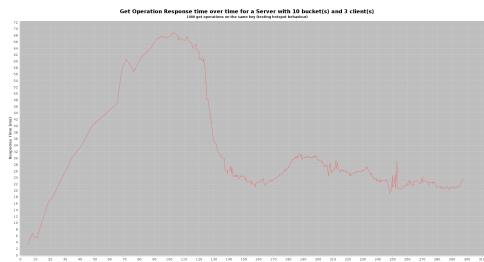


Figura 25: Máquina 1

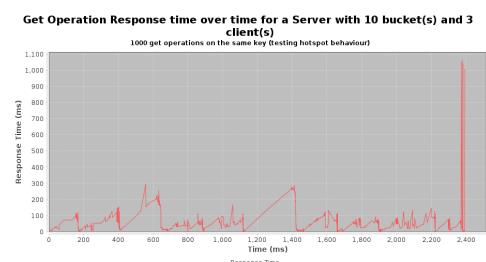


Figura 26: Máquina 2

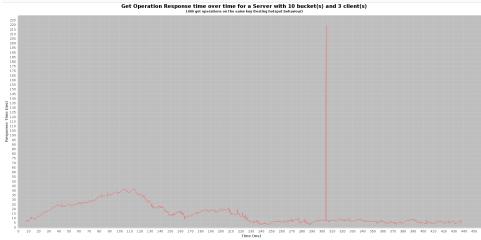


Figura 27: Máquina 3

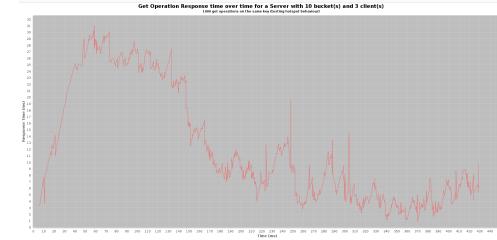


Figura 28: Máquina 4

7.2.3. 100 shards e 10 clientes

A performance começa a degradar-se levemente devido à concorrência e ao maior número de buckets. Podemos observar imensa variância nos tempos de respostas.

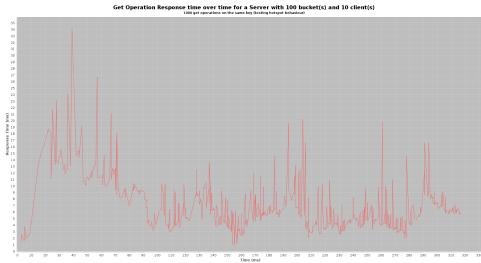


Figura 29: Máquina 1

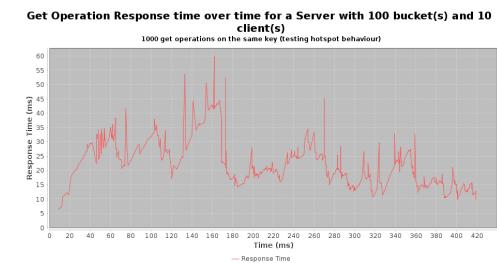


Figura 30: Máquina 2

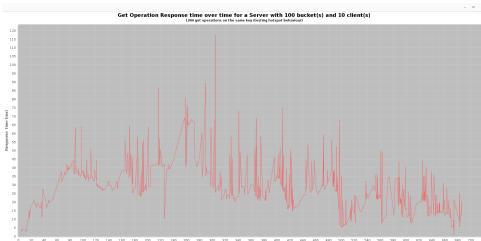


Figura 31: Máquina 3

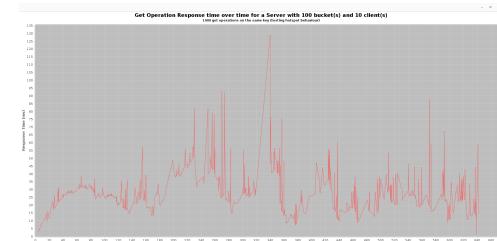


Figura 32: Máquina 4

7.2.4. 1000 shards e 10 clientes

O sistema apresenta tempos de resposta inconsistentes, com picos de latência causados pela sobrecarga de busca num grande número de buckets.

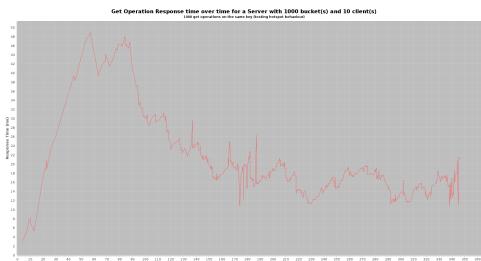


Figura 33: Máquina 1

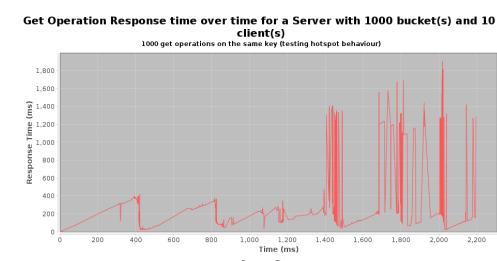


Figura 34: Máquina 2

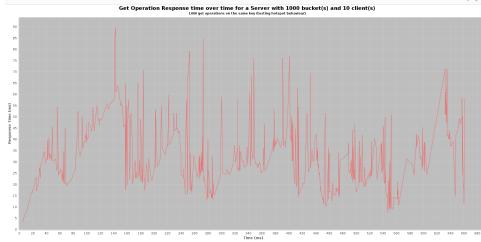


Figura 35: Máquina 3

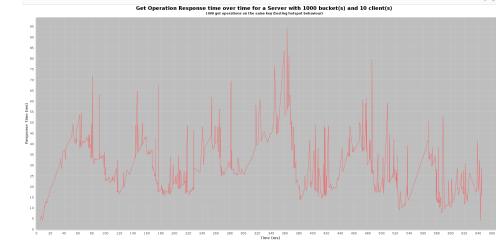


Figura 36: Máquina 4

7.2.5. 1000 shards e 100 clientes

A latência aumenta drasticamente, evidenciando o excesso de concorrência, combinado com um grande número de buckets, cria um *bottleneck* no sistema.

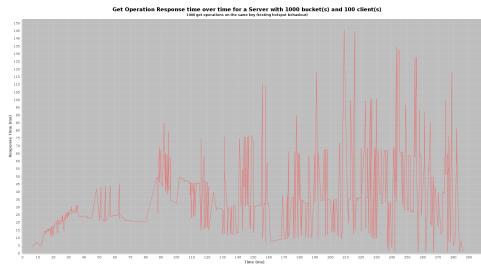


Figura 37: Máquina 1

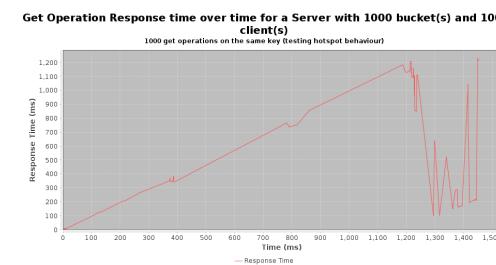


Figura 38: Máquina 2

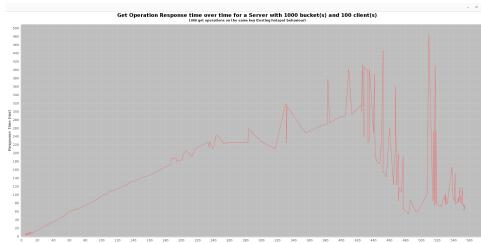


Figura 39: Máquina 3

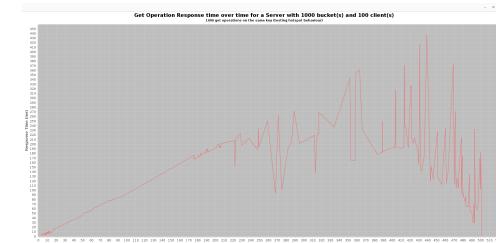


Figura 40: Máquina 4

7.3. MultiGet 1000 pares chave-valor aleatórios

Este workload verifica a capacidade do sistema em lidar com operações em lote em chaves aleatórias.

7.3.1. 10 shards e 1 cliente

O sistema executa bem as operações em lote com baixa sobrecarga, graças à pequena escala de buckets.

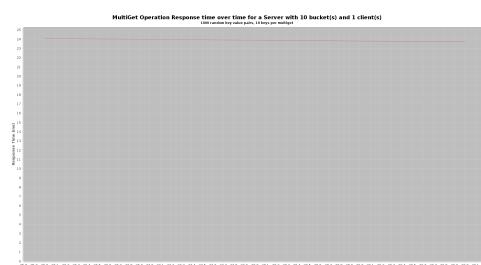


Figura 41: Máquina 1

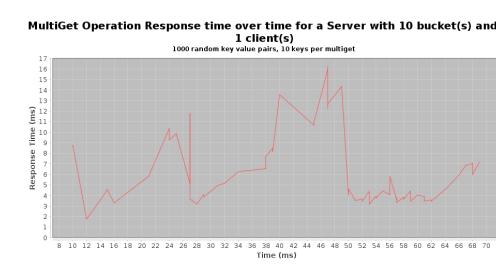


Figura 42: Máquina 2

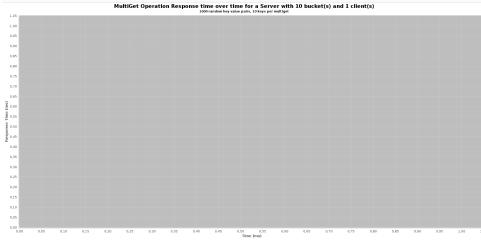


Figura 43: Máquina 3

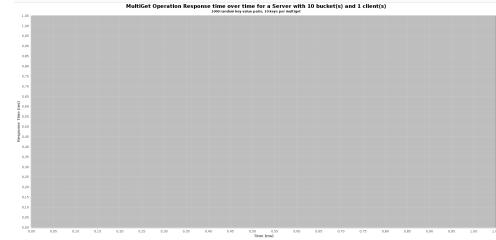


Figura 44: Máquina 4

7.3.2. 10 shards e 3 clientes

O tempo médio de resposta diminuiu ligeiramente devido ao uso de concorrência entre poucos clientes, mas a estabilidade geral é mantida.

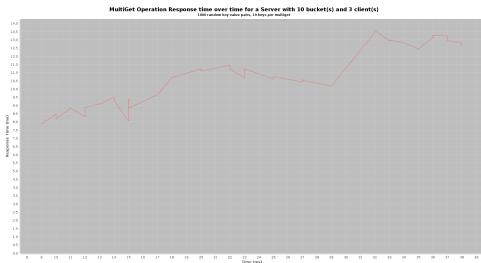


Figura 45: Máquina 1

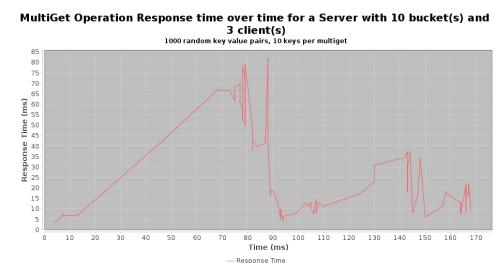


Figura 46: Máquina 2

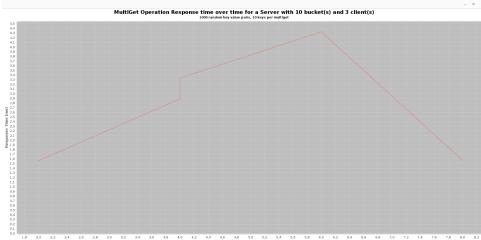


Figura 47: Máquina 3

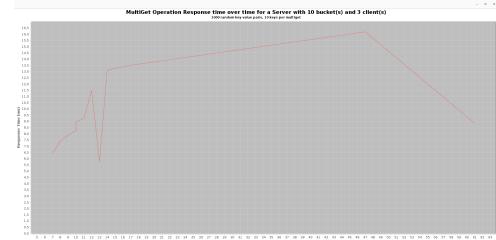


Figura 48: Máquina 4

7.3.3. 100 shards e 10 clientes

A introdução de mais buckets e concorrência resulta numa maior variância nos tempos de resposta, mas o sistema ainda é capaz de processar os lotes.

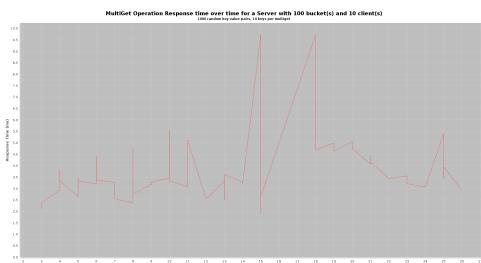


Figura 49: Máquina 1

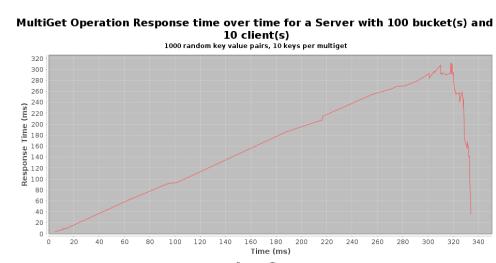


Figura 50: Máquina 2

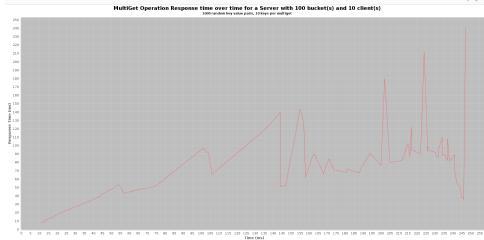


Figura 51: Máquina 3

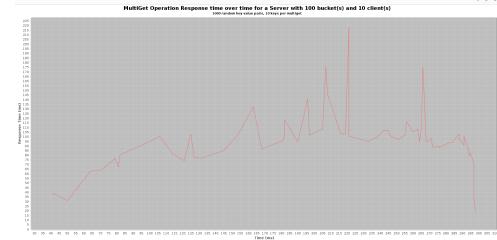


Figura 52: Máquina 4

7.3.4. 1000 shards e 10 clientes

Neste caso, mesmo com a gestão de um elevado número de *buckets* e maior competição de recursos entre os 10 clientes, os tempos de resposta diminuiram, ainda que se verifique uma grande variância consequente dos bloqueios.

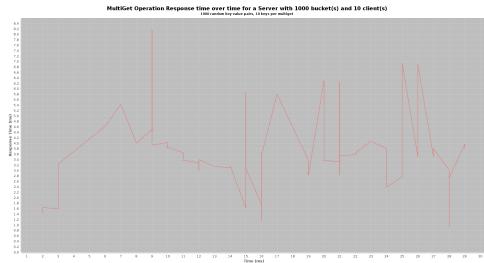


Figura 53: Máquina 1

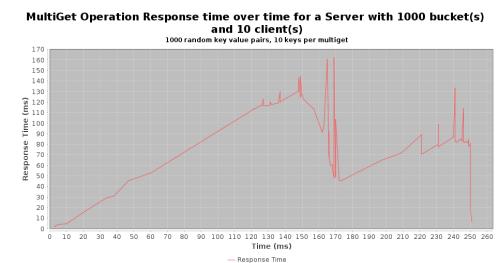


Figura 54: Máquina 2

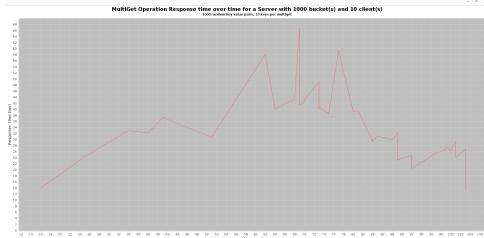


Figura 55: Máquina 3

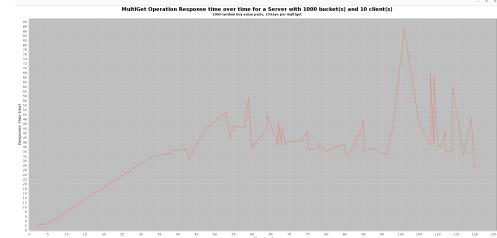


Figura 56: Máquina 4

7.3.5. 1000 shards e 100 clientes

Para este cenário o sistema consegue aproveitar ao máximo a concorrência entre um elevado número de clientes, registando-se tempos de resposta ainda menores.

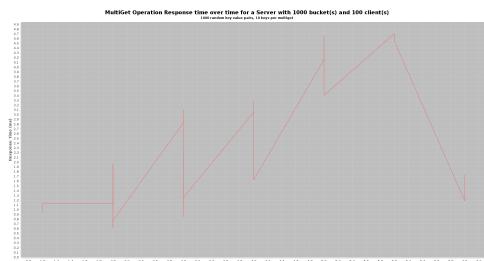


Figura 57: Máquina 1

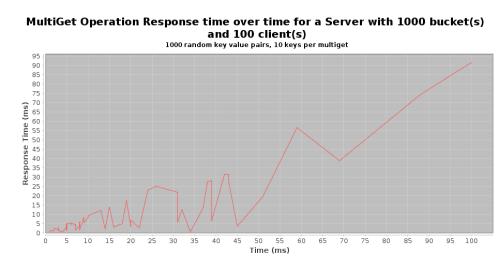


Figura 58: Máquina 2

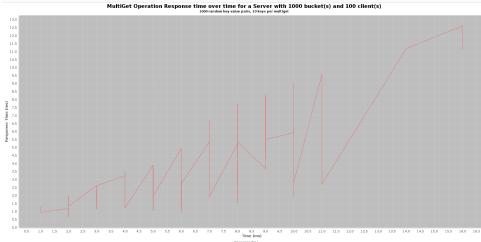


Figura 59: Máquina 3

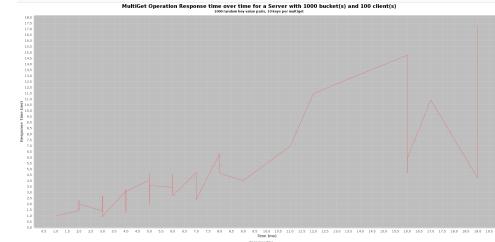


Figura 60: Máquina 4

7.4. MultiGet 100 operações na mesma chave (10 chaves por multiget)

Este workload avalia a eficiência de operações em lote focadas em chaves altamente acedidas.

7.4.1. 10 shards e 1 cliente

A execução em lote é rápida e eficiente, com tempos de resposta baixos devido à simplicidade do cenário. Mais uma vez, o valor do tempo de resposta tende a aumentar ao longo do tempo devido a tratar-se de um processo sequencial (pois é o mesmo cliente a pedir as mesmas chaves repetidamente).

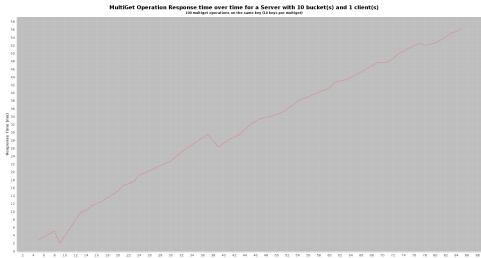


Figura 61: Máquina 1

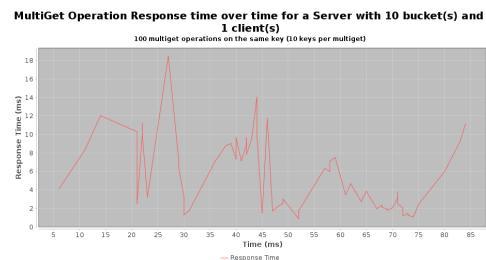


Figura 62: Máquina 2

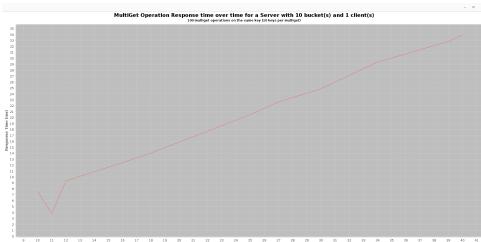


Figura 63: Máquina 3

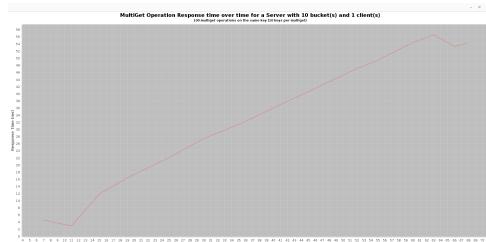


Figura 64: Máquina 4

7.4.2. 10 shards e 3 clientes

Pequenos aumentos na latência são notados, mas o sistema ainda apresenta um desempenho confiável.

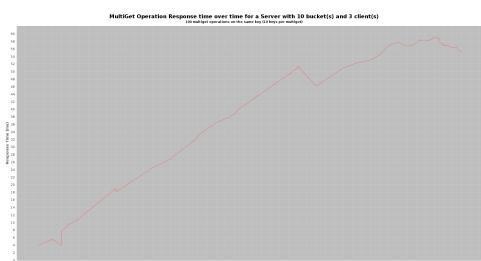


Figura 65: Máquina 1

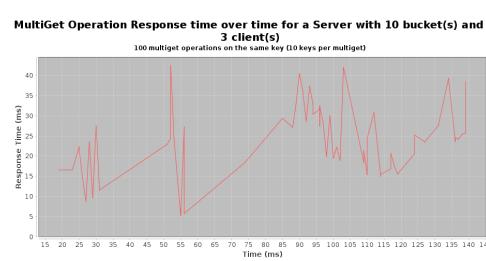


Figura 66: Máquina 2

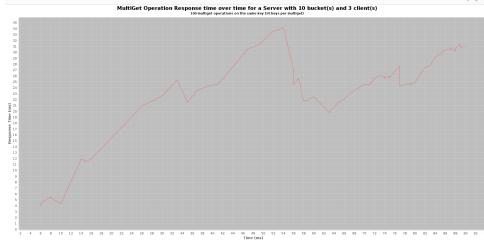


Figura 67: Máquina 3

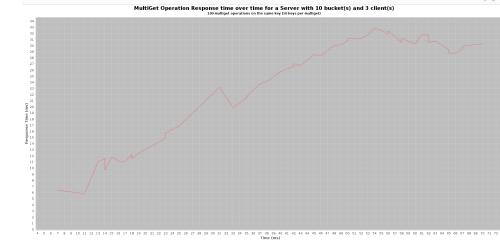


Figura 68: Máquina 4

7.4.3. 100 *shards* e 10 clientes

O aumento do número de *buckets* e clientes a fazer os pedidos inserem maior variância nos tempos de resposta, mas o sistema consegue gerir os recursos de forma mais eficiente.

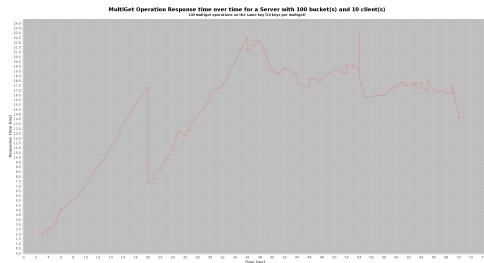


Figura 69: Máquina 1

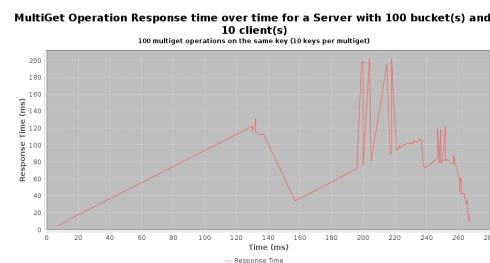


Figura 70: Máquina 2

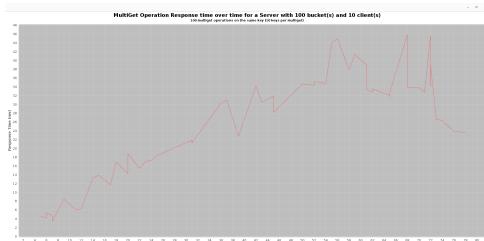


Figura 71: Máquina 3

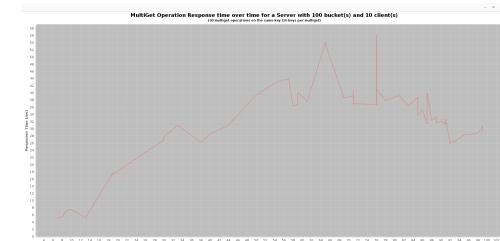


Figura 72: Máquina 4

7.4.4. 1000 *shards* e 10 clientes

O aumento no número de *buckets* gera maior variância nos tempos de resposta, como vimos anteriormente, mas o maior número de clientes faz com que os tempos de resposta sejam ainda menores, devido a um melhor uso da concorrência.

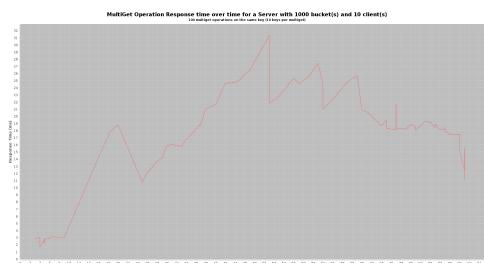


Figura 73: Máquina 1

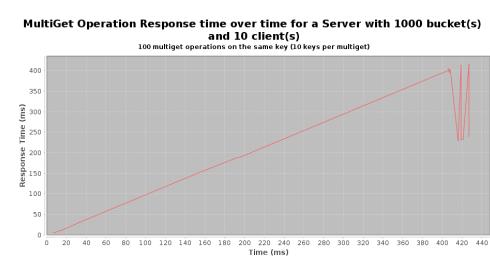


Figura 74: Máquina 2

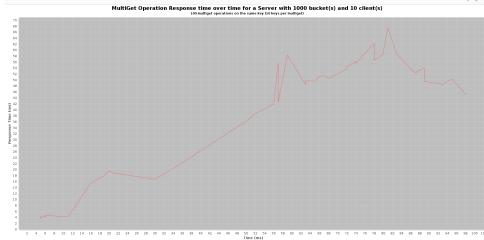


Figura 75: Máquina 3

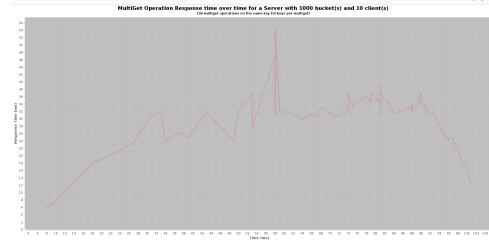


Figura 76: Máquina 4

7.4.5. 1000 shards e 100 clientes

Este cenário é o mais desafiador, mas mesmo com uma gestão de um enorme número de *buckets* e imensos pedidos de clientes diferentes, o sistema consegue atingir os melhores tempos de resposta, ainda que com alguma variância nos valores.

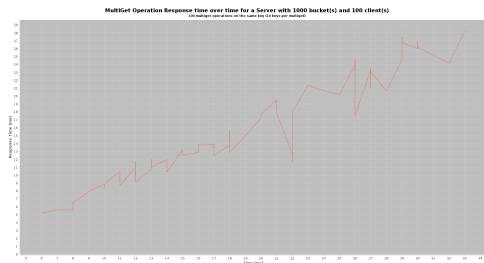


Figura 77: Máquina 1

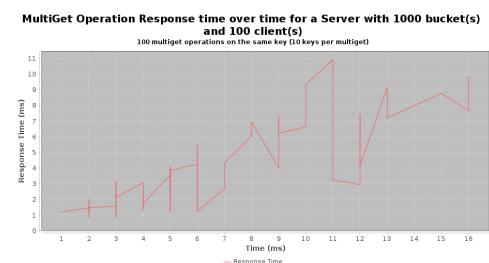


Figura 78: Máquina 2

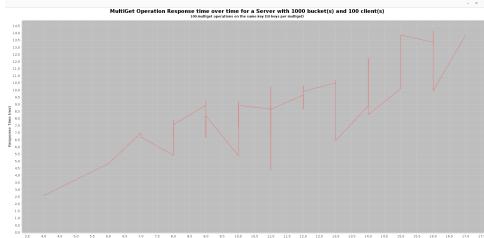


Figura 79: Máquina 3

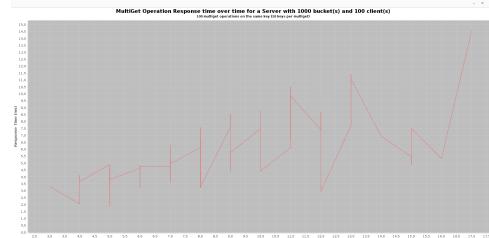


Figura 80: Máquina 4

Os resultados dos testes demonstram que o desempenho do sistema, ainda que um pouco suscetível ao número de *buckets* e clientes, depende principalmente do método utilizado e da máquina em questão, ao contrário do esperado. No entanto, podemos concluir que deve haver um equilíbrio na escolha do número de *buckets* a utilizar e especial atenção à quantidade de pedidos que o sistema pode aceitar de uma vez.

Em suma, podemos concluir que um número muito pequeno de *buckets* gera muita contenção no sistema, aumenta os tempos de resposta e um número muito grande, insere imensa variância nos tempos e sobrecarga adicional na própria gestão dos mesmos. Sobre a concorrência, é claro que executar paralelamente é mais vantajoso do que executar sequencialmente, mas aumentar indefinidamente o número de clientes que o servidor pode processar em simultâneo vai introduzir eventualmente um *bottleneck* na *performance*.

A quantidade e qualidade da memória RAM, bem como a capacidade do processador, desempenham papéis cruciais. Isso é evidenciado pelos gráficos, que mostram variações no desempenho de acordo com esses fatores.

8. Conclusão

Concluindo, acreditamos ter alcançado todos os requisitos apresentados pela equipa docente no enunciado, tendo desenvolvido com sucesso o projeto prático. Este trabalho permitiu-nos melhorar competências na área de sistemas distribuídos e programação concorrente, aplicando efetivamente conceitos lecionados nas aulas, mas também incentivou à exploração de novos assuntos para além das mesmas.

Os testes realizados mostram que o desempenho do sistema, embora sensível ao número de *buckets* e clientes, é influenciado principalmente pelo método utilizado e pelas características da máquina, contrariando as expectativas iniciais.

Conclui-se que é necessário buscar um equilíbrio no número de *buckets*, pois poucos *buckets* aumentam a contenção e os tempos de resposta, enquanto muitos *buckets* introduzem maior variância nos tempos e sobrecarga na gestão. Da mesma forma, embora a execução paralela seja vantajosa, aumentar indiscriminadamente o número de clientes simultâneos pode causar gargalos na *performance*.