



Universidade do Minho
Escola de Engenharia
Licenciatura em Engenharia Informática

Sistemas Distribuídos

Trabalho Prático

Grupo 1



Eduardo Faria
a104353



Hélder Gomes
a104100



Nuno Silva
a104089



Pedro Pereira
a104082

Data da Receção	
Responsável	
Avaliação	
Observações	

Eduardo Faria Hélder Gomes Nuno Silva Pedro Pereira
a104353 a104100 a104089 a104082

28 de dezembro de 2024

Resumo

Este relatório aborda o trabalho prático da Unidade Curricular de [Sistemas Distribuídos](#), inserida no curso de Licenciatura em Engenharia Informática, no ano letivo de 2024/2025. O projeto tem como principal objetivo a implementação de um serviço de armazenamento de dados em memória, com acesso remoto através de uma arquitetura cliente-servidor. O servidor foi concebido para gerir múltiplas conexões simultâneas de clientes, garantindo acesso e manipulação concorrente dos dados.

O sistema foi desenvolvido em Java, utilizando *threads* e sockets *TCP*, com uma estrutura orientada a objetos. A interface de acesso aos dados é baseada em operações de escrita e leitura num formato chave-valor, promovendo uma interação robusta entre clientes e servidor. Estratégias para minimizar a contenção de recursos e otimizar a gestão de *threads* foram adotadas e otimizadas, assegurando um serviço eficiente e escalável. Conta com uma análise de desempenho que inclui cenários de testes úteis e variados, inspirados no *benchmark* de referência YCSB.

Área de Aplicação: Sistemas Distribuídos

Palavras-Chave: Armazenamento em Memória, Cliente-Servidor, Concorrência, Java, Sockets TCP, Arquitetura *Multi-threaded*

Índice

1. Arquitetura do Sistema	1
2. Protocolos do Sistema	2
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. Limite de utilizadores concorrentes	5
3.5. Suporte a clientes <i>multi-threaded</i>	5
3.6. Operação de leitura condicional	5
4. Minimização da Contenção	6
5. Análise dos Resultados dos Testes	9
5.1. Estrutura do Benchmark	9
5.1.1. Workloads Utilizados	9
5.1.2. Configuração do Ambiente de Testes	10
5.1.3. Objetivos	10
5.2. Resultados obtidos	10
5.2.1. Resultados com 100 Operações	10
5.2.2. Resultados com 1.000.000 de Operações	11

Lista de Figuras

Figura 1: Diagrama da arquitetura do sistema	1
Figura 2: Classe respetiva ao pacote AuthPacket.	2
Figura 3: Classe respetiva ao pacote AckPacket.	2
Figura 4: Classe respetiva ao pacote PutPacket.	2
Figura 5: Classe respetiva ao pacote GetPacket.	2
Figura 6: Classe respetiva ao pacote MultiPutPacket.	2
Figura 7: Classe respetiva ao pacote MultiGetPacket.	2
Figura 8: Classe respetiva ao pacote PacketWrapper.	3
Figura 9: Classe respetiva ao pacote GetWhenPacket.	3
Figura 10: Primeira versão da implementação do método update.	6
Figura 11: Segunda versão da implementação do método update.	7
Figura 12: Terceira versão da implementação do método update.	8
Figura 13: Throughput vs Threads - 100 de operações.	11
Figura 14: Average Latency vs Threads - 100 de operações.	11
Figura 15: Throughput vs Threads - 1.000.000 de operações.	12
Figura 16: Average Latency vs Threads - 1.000.000 de operações.	12

1. Arquitetura do Sistema

De modo a simplificar o entendimento da arquitetura do sistema, concebeu-se um diagrama simples representativo da comunicação geral entre os clientes e o servidor.

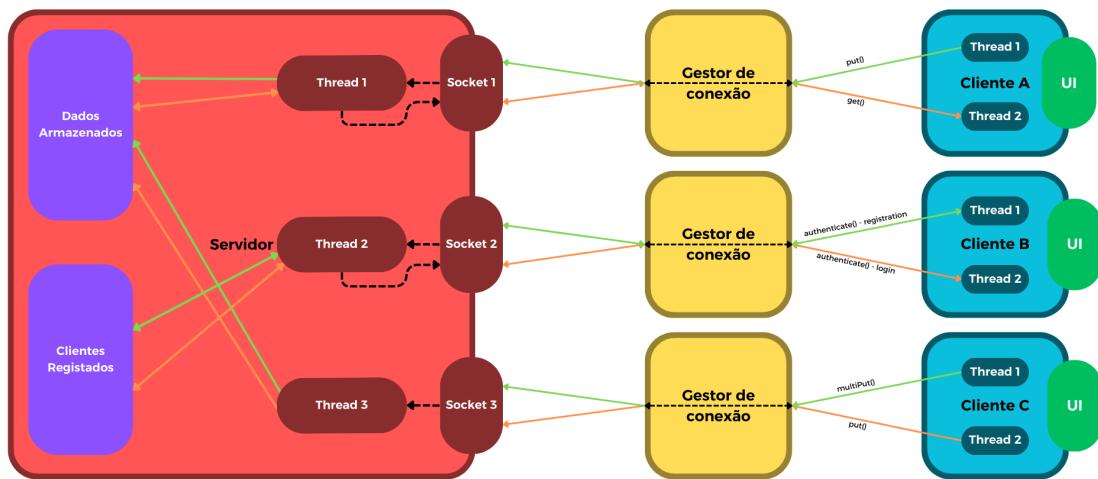


Figura 1: Diagrama da arquitetura do sistema

Um dado utilizador do sistema faz uso da interface para criar ou aceder a uma conta existente. Quando este se autentica, adquire o acesso às funcionalidades do sistema que permitem ao utilizador realizar pedidos ao servidor para armazenar/consultar informação.

O envio de um pedido, e correspondente resposta, quando aplicável, é da responsabilidade do gestor de conexão, que atua como *middleware* na comunicação entre o cliente e o servidor. Este assegura que qualquer solicitação de envio ou receção, tanto pelo servidor como pelo cliente, é concluída de forma simples e uniforme.

A implementação do servidor segue o modelo *thread-per-connection*, instanciando várias *threads* que tratam das conexões estabelecidas pelos clientes. Deste modo, um pedido de um cliente que fique bloqueado no servidor não impede que outros pedidos que outro cliente submeta concorrentemente sejam servidos.

As diferentes *threads* do servidor utilizam, concorrentemente, as estruturas de dados que são responsáveis por manter, em memória, as contas criadas e os pares chave-valor armazenados no sistema.

Um dado utilizador pode realizar um pedido de término do servidor, que resulta num processo de armazenamento do estado do servidor em memória persistente, neste caso, no ficheiro especificado pelo utilizador.

2. Protocolos do Sistema

De forma a garantir uma maior consistência dos dados e aumentar a simplicidade na serialização de informação, optou-se por criar pacotes diferentes com responsabilidades diferentes. Cada um conta com um conjunto distinto de variáveis de instância, bem como um comportamento único, caracterizado pelo forma como são serializados e desserializados.

```
● ● ●  
1 public class AuthPacket {  
2     private String username;  
3     private String password;  
4 }
```

Figura 2: Classe respetiva ao pacote AuthPacket.

```
● ● ●  
1 public class AckPacket {  
2     private boolean ack;  
3 }
```

Figura 3: Classe respetiva ao pacote AckPacket.

```
● ● ●  
1 public class PutPacket {  
2     private String key;  
3     private byte[] data;  
4 }
```

Figura 4: Classe respetiva ao pacote PutPacket.

```
● ● ●  
1 public class GetPacket {  
2     private String key;  
3 }
```

Figura 5: Classe respetiva ao pacote GetPacket.

```
● ● ●  
1 public class MultiPutPacket {  
2     private Map<String, byte[]> pairs;  
3 }
```

Figura 6: Classe respetiva ao pacote MultiPutPacket.

```
● ● ●  
1 public class MultiGetPacket {  
2     private Set<String> keys;  
3 }
```

Figura 7: Classe respetiva ao pacote MultiGetPacket.



```
1 public class PacketWrapper {  
2     private int type;  
3     private Object packet;  
4 }
```



```
1 public class GetWhenPacket {  
2     private String key;  
3     private String keyCond;  
4     private byte[] dataCond;  
5 }
```

Figura 8: Classe respetiva ao pacote PacketWrapper.

Figura 9: Classe respetiva ao pacote GetWhenPacket.

A classe `PacketWrapper` é responsável por atribuir comportamento ao pacote *wrapper* que, por sua vez, encapsula qualquer tipo de pacote, atribuindo-lhe um tipo respetivo. Deste modo, a comunicação bidirecional entre os clientes e o servidor segue um padrão previsível de envio e receção de pacotes. O emissor de um pacote específico assegura que encapsula o mesmo no pacote genérico de modo a que o receptor confira o tipo incluído no pacote genérico e perceba qual o tipo de pacote que deverá tratar.

Os pacotes da classe `AuthPacket` transportam o nome de utilizador e palavra-passe associada para, dependendo do tipo incluído no pacote genérico, ou registrar uma nova conta ou realizar a autenticação numa conta existente.

O servidor faz uso de um pacote da classe `AckPacket` para comunicar ao cliente se uma operação foi bem-sucedida/aceite ou mal-sucedida/rejeitada. Neste caso, este pacote de confirmação é utilizado para comunicar ao cliente se a sua autenticação ou registo foi bem ou mal sucedido.

Os pacotes das classes `PutPacket` e `GetPacket` carregam a informação necessária para a execução dos métodos `void put(String key, byte[] value)` ou `byte[] get(String key)`. Adicionalmente, dado o conteúdo do pacote do tipo `PutPacket`, o mesmo é usado para enviar a informação referente ao par chave-valor requisitado por um pedido de `get()` ou `getWhen()`.

Similarmente ao pacotes singulares, as pacotes das classes `MultiPutPacket` e `MultiGetPacket` carregam a informação necessária para a execução dos métodos `void multiPut(Map<String, byte[]> pairs)` ou `Map<String, byte[]> multiGet(Set<String> keys)`. Adicionalmente, dado o conteúdo do pacote do tipo `MultiPutPacket`, o mesmo é usado para enviar a informação referente aos pares chave-valor requisitados por um pedido de `multiGet()`.

Os pacotes da classe `GetWhenPacket` transportam os dados necessários para execução do método `byte[] getWhen(String key, String keyCond, byte[] valueCond)`.

3. Funcionalidades

Neste capítulo são explicadas as diversas funcionalidades do sistema desenvolvido, onde se destacam os mecanismos de autenticação e registo de clientes, as operações de leitura e escrita, e as funcionalidades que asseguram a concorrência e a consistência na interação entre os clientes e o servidor.

3.1. Autenticação e registo do utilizador

Um dado utilizador pode pedir ao servidor para criar uma conta através de um nome de utilizador e palavra-passe. O servidor responde ao pedido através de um pacote de confirmação (ACK), que carrega um reconhecimento bem-sucedido da operação, ou mal-sucedido caso um mesmo nome de utilizador já se encontre registado.

O utilizador pode ainda pedir para se autenticar no sistema fornecendo um nome de utilizador e uma palavra-passe de acesso. O servidor responde do mesmo modo ao pedido de registo, indicando que a operação correu mal apenas quando o utilizador fornece um nome de utilizador não existente ou uma palavra-passe errada para um dado nome de utilizador. Caso a operação de autenticação seja bem-sucedida, o cliente adquire o acesso às funcionalidades do sistema podendo, a qualquer altura, terminar a sessão na sua conta.

3.2. Operações de escrita e leitura simples

O sistema conta com funcionalidades que permitem a um cliente efetuar pedidos de registo de informação associada a uma dada chave, através do método `void put(String key, byte[] value)`, ou pedidos de leitura de informação associada a uma dada chave, com o método `byte[] get(String key)`. Os métodos exercidos por parte do cliente limitam-se a fazer um pedido de envio de pacote ao gestor de conexão, que remete o envio ao servidor, e realizar um pedido de receção de informação ao gestor de conexão no caso do método `get()`.

O servidor, quando recebe um dos dois pedidos, faz uso das suas estruturas de dados para cumprir a ação solicitada pelo cliente. A implementação dos métodos responsáveis por atuar sobre estes pedidos foi feita de modo a assegurar exclusão mútua e minimizar a contenção dos mecanismo de *locks* na zona crítica. Um exemplo da implementação do método `update()` — método responsável por pedidos `put()` dos clientes — pode ser visto no [capítulo 4](#).

3.3. Operações de escrita e leitura compostas

O sistema conta com funcionalidades que permitem a um cliente efetuar pedidos de registo atómico de vários dados associados a várias chaves, através do método `void multiPut(Map<String, byte[]> pairs)`, ou pedidos de leitura de vários dados associados a várias chaves, com o método `Map<String, byte[]> multiGet(Set<String> keys)`. Os métodos exercidos por parte do cliente limitam-se a fazer um pedido de envio de pacote ao gestor de conexão, que remete o envio ao servidor, e realizar um pedido de receção de informação ao gestor de conexão no caso do método `multiGet()`.

O servidor, quando recebe um dos dois pedidos, faz uso das suas estruturas de dados para cumprir a ação solicitada pelo cliente. A implementação dos métodos responsáveis por atuar sobre estes pedidos foi feita de modo a assegurar exclusão mútua e minimizar a contenção dos mecanismo de *locks* na zona crítica. A implementação destes métodos itera por todos os elementos das estruturas de dados passadas como argumento e realiza procedimentos similares às operações singulares.

3.4. Limite de utilizadores concorrentes

O servidor tem acesso a um parâmetro “S” que é passado como argumento na inicialização do servidor e é responsável por limitar o número sessões concorrentes (diferentes clientes a usar o servidor). Para além disso, o servidor contabiliza o número de clientes que estão atualmente conectados, tendo, para isso, uma variável que incrementa sempre que um cliente se autentica no servidor após estabelecer conexão, ou seja, sempre que uma sessão é aberta. Quando o número de sessões atinge o valor “S”, qualquer outra tentativa de autenticação por parte de outros clientes é bloqueada, ficando estes à espera de uma resposta de “autorização” do servidor. Sempre que um cliente termina a sua conexão com o servidor, é enviado um sinal para acordar uma das *threads* do servidor adormecidas no processo de autenticação de um dos clientes e, caso a condição “número de sessões abertas é menor do que S” seja cumprida, a autenticação do cliente prossegue até ser enviada a confirmação de volta ao mesmo.

3.5. Suporte a clientes *multi-threaded*

Os clientes foram desenvolvidos com uma arquitetura *multi-threaded*, permitindo a execução simultânea de várias tarefas de forma não bloqueante, como pedidos de leitura, escrita e leitura condicional. Esta abordagem aumenta significativamente a eficácia, o desempenho e a responsividade do sistema, uma vez que cada operação pode ser tratada em paralelo. Por exemplo, um cliente que realiza um pedido condicional pode continuar a criar e processar novos pedidos, mesmo que o pedido condicional permaneça pendente, algo que seria inviável numa arquitetura *single-threaded*. Esta implementação assegura um processamento contínuo e eficiente, eliminando bloqueios e otimizando a interação entre o cliente e o sistema.

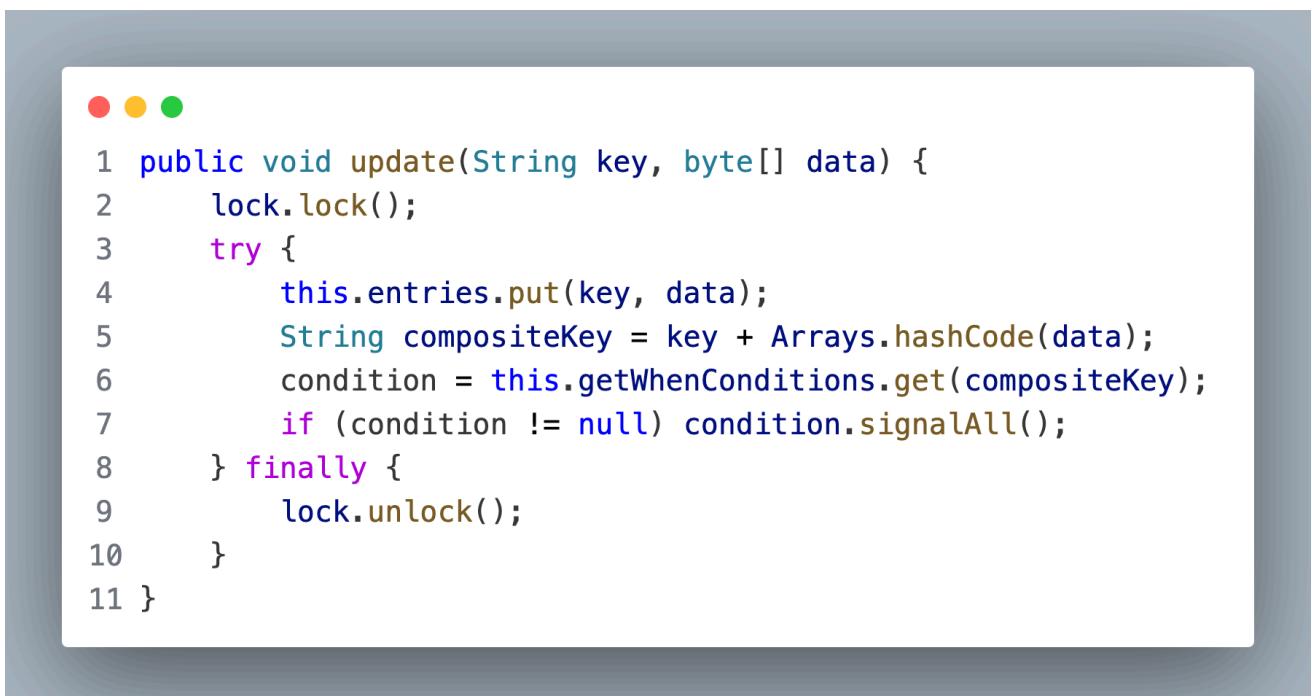
3.6. Operação de leitura condicional

Cada cliente pode indicar ao servidor que pretende receber os dados associados a uma dada chave quando a informação relativa a uma “chave de condição” coincide com a “informação de condição” pretendida, a partir do método `byte[] getWhen(String key, String keyCond, byte[] valueCond)`. Neste caso, a *thread* do servidor (*thread* da conexão) responsável por operar este pedido ficará adormecida enquanto que a informação pretendida ainda não coincide com a informação esperada pelo cliente. Para este efeito, é criada, caso não exista, uma condição unicamente respetiva a uma chave resultante da concatenação entre a string “keyCond” e a *hash* de “valueCond”. Posteriormente, esta é usada de forma a garantir que as únicas *threads* que são acordadas por um sinal (*signalAll*) são aquelas que estão adormecidas pela condição respetiva a esta chave composta. Desta forma, sempre que alguma informação é inserida ou atualizada pelo servidor, é enviado um sinal a partir da condição respetiva, caso exista.

4. Minimização da Contenção

A minimização da contenção dos *locks* foi uma etapa crucial na modificação das implementações existentes. Esta modificação passou por um processo iterativo, exemplificado no método *update* (servidor) nas seguintes imagens.

Numa primeira versão, o método fazia uso de um *lock* genérico para aceder às duas estruturas de dados existentes no servidor — mapa de *entries* e mapa de *getWhenConditions*.



The screenshot shows a Java code editor with a dark theme. At the top left are three circular icons: red, yellow, and green. The code itself is as follows:

```
1 public void update(String key, byte[] data) {  
2     lock.lock();  
3     try {  
4         this.entries.put(key, data);  
5         String compositeKey = key + Arrays.hashCode(data);  
6         condition = this.getWhenConditions.get(compositeKey);  
7         if (condition != null) condition.signalAll();  
8     } finally {  
9         lock.unlock();  
10    }  
11 }
```

Figura 10: Primeira versão da implementação do método *update*.

Após uma revisão cuidadosa da implementação, evidenciou-se que a linha 4, 6 e 7 eram referentes a duas estruturas de dados diferentes. Adicionalmente, a linha 5 não necessitava de mais nenhuma informação dos mapas, logo, não precisaria de estar inserida num mecanismo de *lock*.

Concebeu-se então uma segunda iteração da implementação que visava utilizar *locks* mais granulares, referentes às estruturas de dados a que competem. As linhas que não requeressem de um acesso a uma zona crítica foram colocadas fora de qualquer mecanismo de *lock*.

```

1 public void update(String key, byte[] data) {
2     entriesLock.lock();
3     try {
4         this.entries.put(key, data);
5     } finally {
6         entriesLock.unlock();
7     }
8
9     String compositeKey = key + Arrays.hashCode(data);
10
11    Condition condition;
12    whenConditionsLock.lock();
13    try {
14        condition = this.getWhenConditions.get(compositeKey);
15    } finally {
16        whenConditionsLock.unlock();
17    }
18
19    entriesLock.lock();
20    try {
21        if (condition != null) condition.signalAll();
22    } finally {
23        entriesLock.unlock();
24    }
25 }
```

Figura 11: Segunda versão da implementação do método update.

De forma a aumentar ainda mais a eficiência do programa, caracterizaram-se as responsabilidades de cada mecanismo de *lock* entre *lock* de escrita ou *lock* de leitura. Para o efeito, a definição dos *locks* passou a derivar da classe [ReentrantReadWriteLock](#).

```
 1 public void update(String key, byte[] data) {
 2     writeEntriesLock.lock(); // write lock
 3     try {
 4         this.entries.put(key, data);
 5     } finally {
 6         writeEntriesLock.unlock();
 7     }
 8
 9     String compositeKey = key + Arrays.hashCode(data);
10
11    Condition condition;
12    readWhenConditionsLock.lock(); // read lock
13    try {
14        condition = this.getWhenConditions.get(compositeKey);
15    } finally {
16        readWhenConditionsLock.unlock();
17    }
18
19    writeEntriesLock.lock(); // write lock
20    try {
21        if (condition != null) condition.signalAll();
22    } finally {
23        writeEntriesLock.unlock();
24    }
25 }
```

Figura 12: Terceira versão da implementação do método update.

Nesta versão final, o uso dos *locks* está bastante granular, e cada acesso a uma zona crítica é feito apenas quando adquirido o *lock* referente à estrutura de dados e tipo de acesso pretendido.

5. Análise dos Resultados dos Testes

Nesta secção, apresenta-se a metodologia adotada pelo grupo de trabalho para realizar testes de desempenho e esforço do sistema desenvolvido. Para tal, foram conduzidos testes de benchmarking, com base na ferramenta YCSB (Yahoo! Cloud Serving Benchmark), uma réplica adaptada do serviço original fornecido pela Yahoo!, amplamente utilizada para avaliar sistemas de armazenamento e bases de dados.

Os testes de benchmarking consistem em avaliar o desempenho do sistema em cenários representativos, simulados através de **workloads** específicos. Estes workloads reproduzem padrões típicos de utilização do sistema, possibilitando a medição de métricas chave de desempenho como **latência**, **throughput**, e o **tempo total de execução**.

5.1. Estrutura do Benchmark

Os resultados obtidos durante os testes foram registados num ficheiro CSV com o seguinte formato de cabeçalho:

```
Workload;Threads;Operations;Duration(ms);AvgLatency(ms);Throughput(ops/sec)
```

- **Workload:** Representa o tipo de operação testada (**Put**, **Get**, **MixedPutGet**, entre outros), indicando a proporção de operações de escrita e leitura realizadas.
- **Threads:** Número de threads concorrentes a executar operações, simulando utilizadores simultâneos.
- **Operations:** Número total de operações realizadas durante o teste.
- **Duration (ms):** Duração total do teste em milissegundos.
- **AvgLatency (ms):** Latência média por operação, calculada com base no tempo de resposta individual.
- **Throughput (ops/sec):** Número de operações concluídas por segundo, medindo a capacidade do sistema sob a carga aplicada.

5.1.1. Workloads Utilizados

Os workloads utilizados refletem diferentes cenários de uso para compreender o comportamento do sistema sob diferentes condições:

1. **WorkloadPut-100W:** Todas as operações consistem em escritas (**Write-Only**), avaliando o desempenho em situações de inserção de dados intensiva.
2. **WorkloadGet-100R:** Todas as operações consistem em leituras (**Read-Only**), ideal para avaliar a capacidade de recuperação de dados.
3. **WorkloadMultiPut-100W:** Semelhante ao **WorkloadPut-100W**, mas cada operação pode envolver múltiplos pedidos de escrita.
4. **WorkloadMultiGet-100R:** Semelhante ao **WorkloadGet-100R**, mas cada operação pode retornar múltiplos resultados.
5. **WorkloadGetWhen:** Testa operações condicionais, verificando a capacidade do sistema de realizar consultas de acordo com o critério indicado.
6. **WorkloadMixedPutGet-<X>Get:** Combinação de operações de leitura e escrita, com proporções ajustadas (por exemplo: 70% leituras, 30% escritas).
7. **WorkloadMixedMultiPutMultiGet-<X>Get:** Combinação de múltiplas operações de leitura e escrita por transação, simulando cenários complexos.

5.1.2. Configuração do Ambiente de Testes

Os testes foram realizados variando o número de threads (1, 5, 10, 20 e 50), permitindo avaliar o impacto da concorrência no desempenho do sistema. O número total de operações foi inicialmente mantido em 100 para os testes preliminares e ampliado para 1.000.000 nas avaliações de escalabilidade. Durante todas as execuções, o sistema foi monitorizado de forma rigorosa para recolher métricas detalhadas de desempenho.

Adicionalmente, todos os workloads foram configurados com probabilidades induzidas de execução, garantindo uma aproximação realista aos padrões de utilização esperados no sistema. Para facilitar esta configuração, foi disponibilizada uma classe dedicada para ajustar os workloads e os respetivos pesos, permitindo personalizar a proporção das operações realizadas em cada teste e refletir cenários mais próximos do contexto real de utilização.

5.1.3. Objetivos

- Determinar a capacidade de escalabilidade do sistema sob diferentes níveis de carga.
- Avaliar a eficiência em cenários de leitura, escrita e mistos.
- Identificar potenciais limitações no desempenho.

5.2. Resultados obtidos

5.2.1. Resultados com 100 Operações

Na primeira fase de testes, foram realizadas 100 operações enquanto o número de threads variava entre 1, 5, 10 e 20. Estes testes foram desenhados para observar o comportamento do sistema em cenários de baixa carga e com um conjunto limitado de threads.

Os resultados mostraram que o throughput (operações por segundo) aumentava proporcionalmente com o número de threads até certo ponto, evidenciando a escalabilidade inicial do sistema. Contudo, foram observadas variações significativas na latência média em workloads com operações mistas, o que sugere que o aumento de concorrência impacta negativamente a consistência do desempenho. Workloads específicos, como o WorkloadPut-100W, apresentaram melhor desempenho, enquanto workloads mistos tiveram uma distribuição mais dispersa em termos de latência.

Este cenário de baixa carga evidenciou que o sistema é sensível à configuração do workload e à complexidade das operações, confirmando a possibilidade de necessidade de ajustes para cenários mais desafiadores.

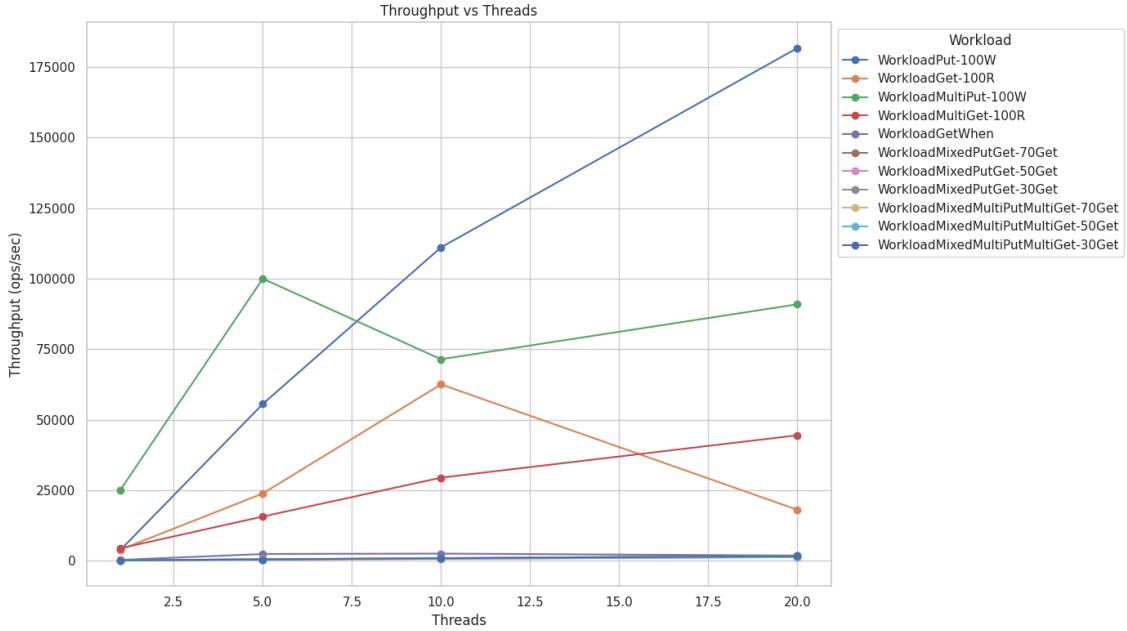


Figura 13: Throughput vs Threads - 100 de operações.

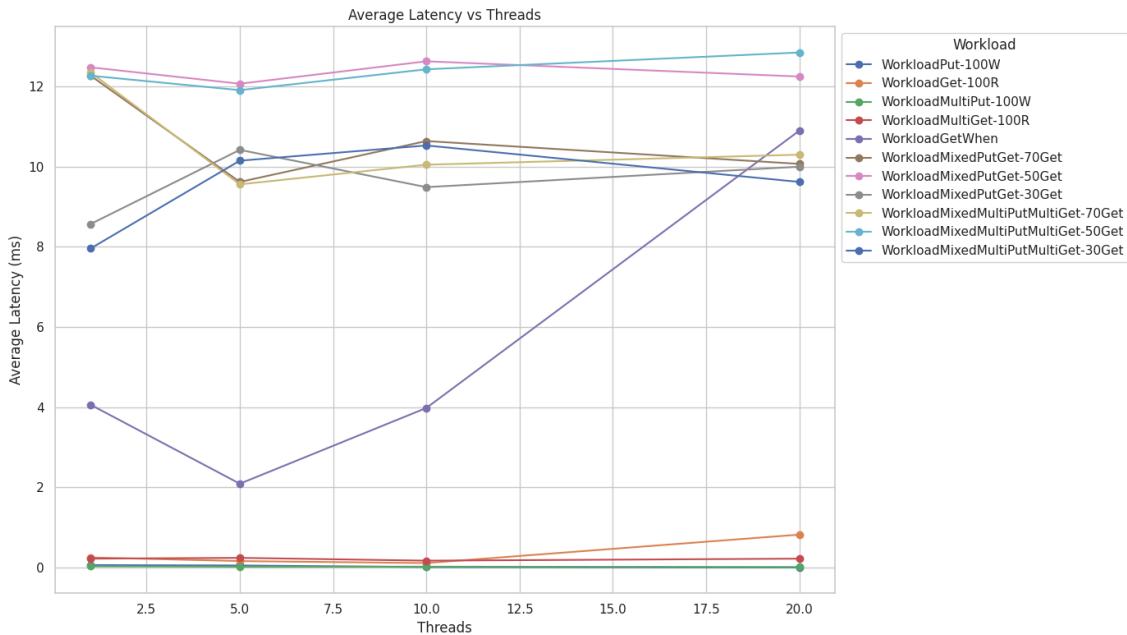


Figura 14: Average Latency vs Threads - 100 de operações.

5.2.2. Resultados com 1.000.000 de Operações

Nos testes de escalabilidade, realizados com 1.000.000 de operações e um número mais amplo de threads (1, 5, 10, 20 e 50), os resultados indicaram tendências claras tanto em termos de throughput quanto de latência.

O gráfico “Throughput vs Threads” revela que workloads como o WorkloadPut-100W escalaram significativamente, atingindo mais de 3 milhões de operações por segundo com 50 threads. Por outro lado, workloads mistos, como o WorkloadMixedPutGet-50Get, apresentaram um crescimento mais moderado, sugerindo que a complexidade das operações afeta a capacidade de processar um número elevado de requisições simultâneas.

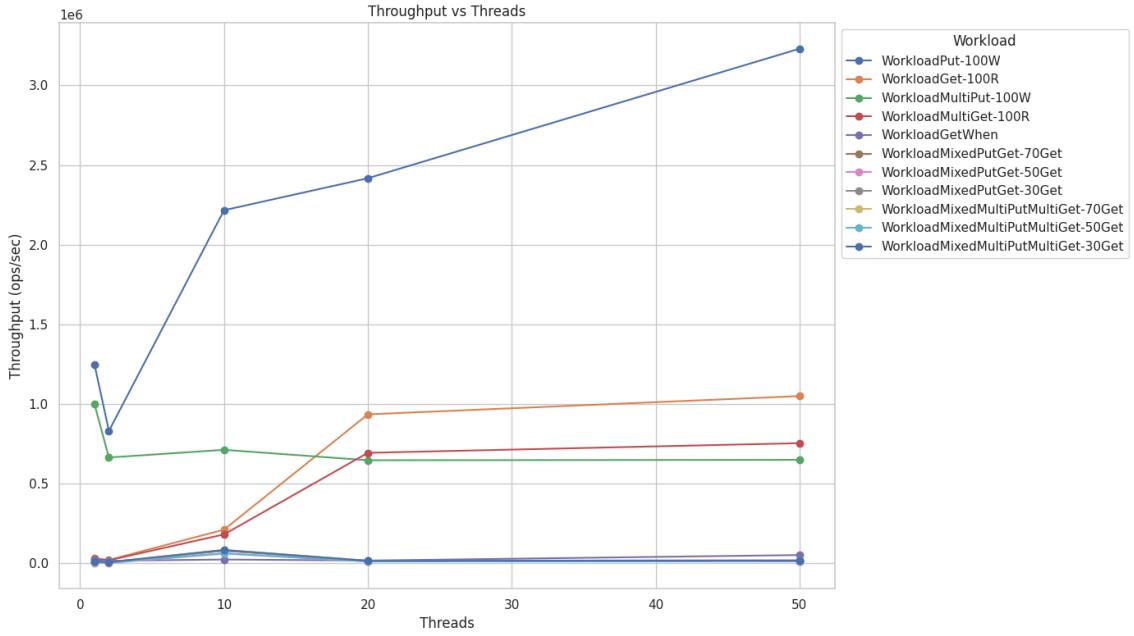


Figura 15: Throughput vs Threads - 1.000.000 de operações.

O gráfico “Average Latency vs Threads” evidencia que a latência média para workloads mais simples, como o WorkloadGet-100R, permaneceu relativamente estável, enquanto workloads mistos apresentaram oscilações com o aumento de threads. Por exemplo, a latência média do WorkloadMixedPutGet-70Get reduziu com o aumento de threads até certo ponto, mas estabilizou em níveis relativamente baixos.

Estes resultados sugerem que o sistema é capaz de lidar eficientemente com operações isoladas em alta concorrência, mas a presença de operações mistas pode induzir limitações que afetam tanto o throughput quanto a latência. Estes cenários evidenciam a importância de otimizar o sistema para workloads heterogêneos, especialmente em ambientes altamente concorrentes.

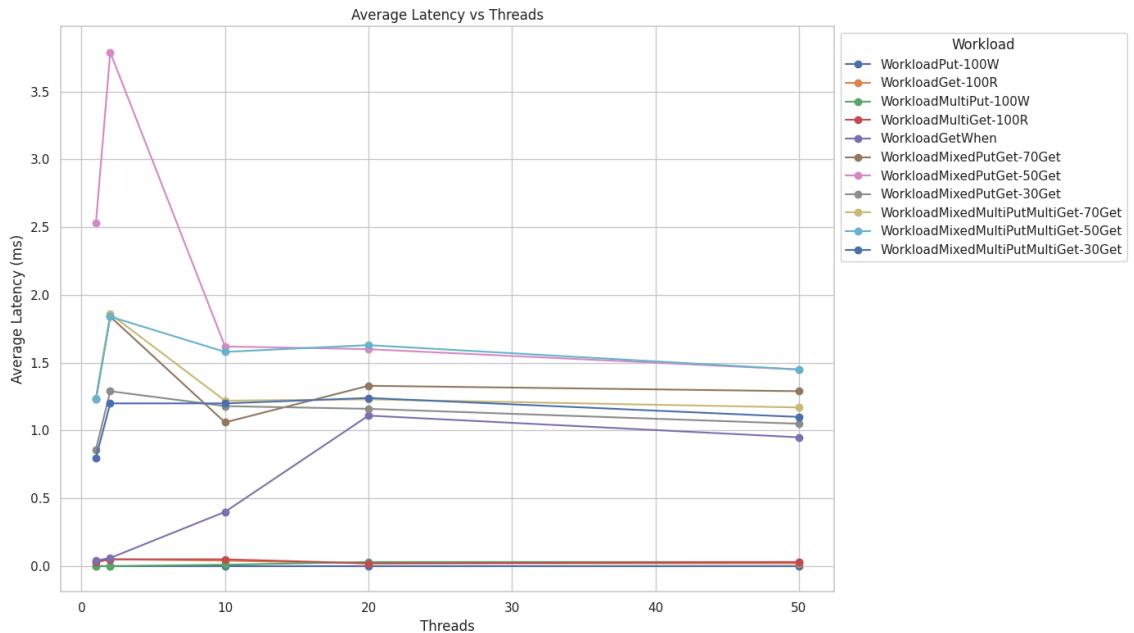


Figura 16: Average Latency vs Threads - 1.000.000 de operações.