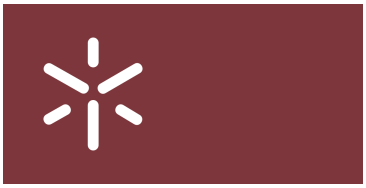

Universidade do Minho
Escola de Engenharia
Licenciatura em Engenharia Informática
Mestrado Integrado em Engenharia Informática

Unidade Curricular de Comunicações por Computador

Ano Letivo de 2024/2025

Armazenamento de dados em memória com acesso remoto

André Miranda a104088
Diogo Outeiro a104092
José Soares a103995
Nuno Melo 104446

Dezembro, 2024

SD

Data da Receção	
Responsável	
Avaliação	
Observações	

Armazenamento de dados em memória com acesso remoto

André Miranda a104088
Diogo Outeiro a104092
José Soares a103995
Nuno Melo 104446

Dezembro, 2024

Resumo

<<Neste projeto, o objetivo é implementar um serviço de armazenamento de dados partilhado, em que a informação é mantida num servidor e acedida remotamente. Os clientes interagem com o servidor por sockets TCP, de forma a inserir e consultar informação.O servidor atende os pedidos do ciente concorrentemente e armazena a informação em memória.A informação armazenada é do tipo chave-valor.Tivemos ainda o cuidado de encontrar estratégias que diminuam a contenção e o número de threads. Neste relatório técnico será explicado ao detalhe a arquitetura e protocolos do sistema decididos pelo grupo. Na fase final do projeto, foram criados diferentes cenários de testes incluindo cargas de trabalho com diferentes tipos de operações e testes de escalabilidade com o intuito de compreender o desempenho de serviço.>>

Área de Aplicação: <<Serviço de Armazenamento de Dados Partilhados>>

Palavras-Chave: <<Conjunto de palavras-chave que permitirão referenciar domínios de conhecimento, tecnologias, estratégias, etc., directa ou indirectamente referidos no relatório. Por exemplo: Bases de Dados Relacionais, Gestão de Índices, JAVA, Protocolos de Comunicação.>>

Índice

1. Introdução **1**

2. Arquitetura do código **2**
2.1. Servidor 3
2.2. Protocolos e Comunicação 3
2.3. Estrutura de Armazenamento 3
2.4. Execução de Operações (Requests) 3

3. Implementação **4**
3.1. Funcionalidades Implementadas 4
3.2. Estrutura do Código 4
3.3. Decisões de Projeto 5

4. Testes e resultados **6**
4.1. Cenários de Teste 6
4.1.1. Métricas Avaliadas 6
4.2. Resultados 7
4.2.1. Teste em Big Set com número de threads variável 7
4.2.2. Teste em Small Set com número de threads variável 8
4.2.3. Comparações finais 9

5. Conclusões e Trabalho Futuro **10**

Lista de Figuras

Figura 1: Arquitetura do Sistema 2
Figura 2: Read Heavy Workload Results 7
Figura 3: Write Heavy Workload Results 7
Figura 4: Balanced Workload Results 7
Figura 5: Read Heavy Workload Results 8
Figura 6: Write Heavy Workload Results 9
Figura 7: Balanced Workload Results 9

1. Introdução

Neste relatório, descrevemos o desenvolvimento de um sistema de armazenamento de dados chave-valor distribuído, incluindo a sua concepção, implementação, e validação. Este projeto teve como objetivo explorar técnicas de concorrência e distribuição, garantindo consistência e alta disponibilidade. O sistema foi projetado para suportar múltiplos clientes simultâneos, mantendo um desempenho robusto e comportando-se corretamente em cenários de carga elevada.

2. Arquitetura do código

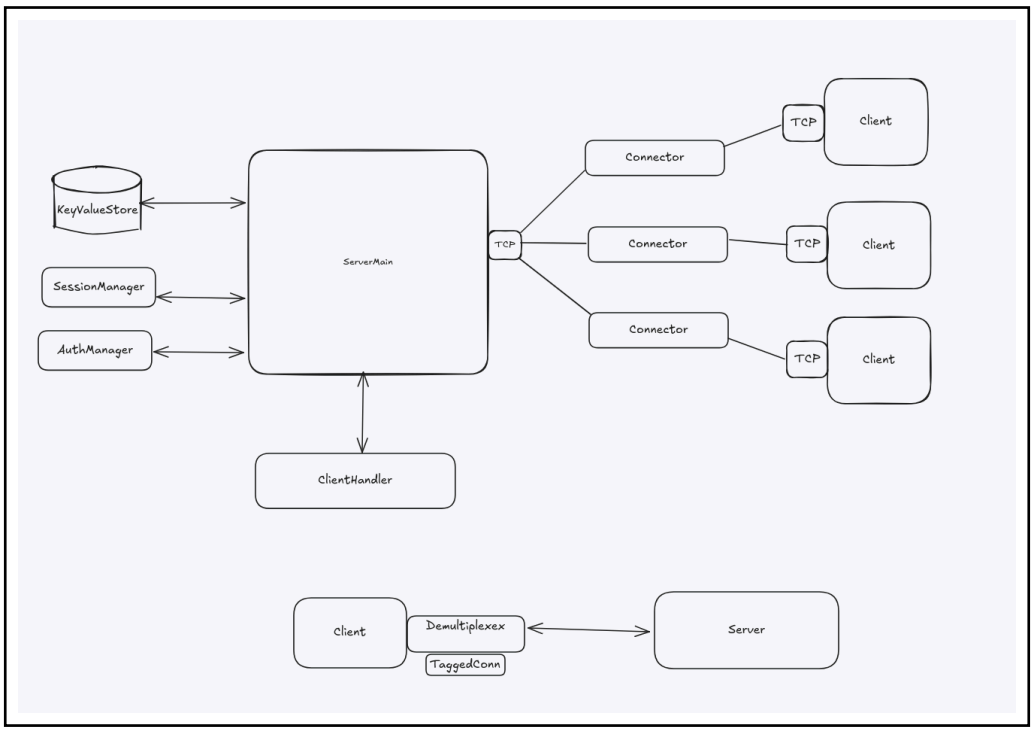

Figura 1: Arquitetura do Sistema

A arquitetura do código foi organizada em componentes principais:

Servidor: É responsável por gerir o armazenamento e processar operações de leitura e escrita. Inclui mecanismos de sincronização para suportar múltiplos clientes.

ClienteInterface e ClientAPI: Juntos formam uma interface simples para que os utilizadores interajam com o sistema, capaz de enviar comandos de leitura e escrita, que são processados pela API.A interface mostra as opções com as operações que o servidor pode oferecer e executar no serviço de armazenamento. Depois a resposta será enviada ao respetivo cliente.

Demultiplexer: Os pedidos dos clientes são armazenados em uma estrutura de dados semelhante a uma queue para depois serem executados concorrentemente pelas diferentes threads. Ainda está presente no demultiplexer a TaggedConnection que associa os pedidos com as devidas respostas.

Common Utilities: Contém as ferramentas comuns, como implementações do armazenamento chave-valor e mecanismos de sincronização, que são usados por diferentes partes do sistema.Nesta parte do projeto, está presente a estrutura de dados que guarda a informação do tipo chave-valor e ainda uma estratégia de ThreadPool.Os pedidos são guardados numa queue e para cada um deles é criada uma thread resposnsável por executá-la. Quando a queue encontra-se vazia, as threads aguardam e caso algum pedido chegue ao servidor estas serão acordadas.

Protocol: Este módulo contem todas as estruturas e ferramentas opacas que são necessárias para a conexão servidor-cliente.

2.1. Servidor

O servidor foi implementado para gerir operações concorrentes de leitura e escrita em um armazenamento chave-valor. Foram usados locks granulares para assegurar que diferentes chaves podem ser acedidas simultaneamente, evitando bloqueios desnecessários. A versão final utiliza Read-Write Locks para melhorar o desempenho em cenários de leitura intensiva.

Funcionalidades principais do servidor incluem:
- Operação PUT: Permite que os clientes insiram ou atualizem valores associados a uma chave.
- Operação GET: Recupera valores armazenados para uma chave específica.
- Operação MultiPut e MultiGet: Executa operações compostas, sendo que mantém todos os requisitos aplicados às operações singulares.
- Operação GetWhen: Implementa uma espera condicional para sincronização de valores entre chaves.

2.2. Protocolos e Comunicação

A comunicação entre cliente e servidor utiliza sockets TCP, com um protocolo binário implementado através das classes TaggedConnection e TaggedFrame. Os dados são organizados em quadros etiquetados para identificar o tipo de operação e garantir a atomicidade. A TaggedConnection classe especializa a TaggedFramed,onde o conteúdo de cada frame contem uma etiqueta (inteiro) seguida de dados.

A etiqueta 1 equivale a um pedido de registo, o 2 equivale a uma escrita, o 3 a uma leitura,o 4 a uma escrita composta, o 5 a uma leitura composta e o 6 a uma leitura condicional.

2.3. Estrutura de Armazenamento

Inicialmente, utilizámos uma KeyValueStore, onde um único bloqueio era responsável por gerir o acesso a toda a store, o que garantiu simplicidade no controlo de concorrência. No entanto, à medida que o número de operações e de chaves aumentava, esta abordagem revelou-se ineficiente, uma vez que o bloqueio global causava contenções – ou seja, múltiplos processos ou threads tinham de esperar para aceder à store, o que resultava numa degradação de desempenho. Para resolver esta limitação, migrámos para uma KeyValueStore com bloqueio por chave (lock per key), uma abordagem mais granular. Com esta alteração, cada chave passou a ter o seu próprio bloqueio e permitiu que acessos simultâneos a chaves diferentes ocorressem sem interferência, melhorando a concorrência e o desempenho. Esta solução demonstrou ser mais eficiente em cenários com um grande número de chaves, pois reduziu o overhead causado pelo bloqueio global e distribuiu de forma mais eficiente o controlo de acesso entre as várias chaves da store.

2.4. Execução de Operações (Requests)

A ThreadPool no servidor tem como principal objetivo otimizar o processamento de múltiplas requisições simultâneas com maior eficiência, desempenho e escalabilidade. É composta por um conjunto de threads de trabalho (workers), responsáveis por executar as tarefas enviadas pelos clientes. A orquestração dessas tarefas é feito através de uma fila de requisições (RequestQueue), onde as tarefas ficam armazenadas até que uma thread esteja disponível para processá-las. Quando uma thread de trabalho é libertada, retira a próxima tarefa da fila e executa.

3. Implementação

3.1. Funcionalidades Implementadas

O sistema implementa uma série de funcionalidades essenciais e avançadas, que foram cuidadosamente projetadas e organizadas em pacotes para facilitar a manutenção e a extensibilidade do código:

Autenticação e Registro de Utilizador: Gerenciada pela classe AuthManager, esta funcionalidade permite que utilizadores se registrem e se autentiquem utilizando um nome de utilizador e uma senha. Essa implementação utiliza locks para garantir que as operações no base de dados de utilizadores sejam thread--safe, evitando condições de corrida.

Operações Simples de Escrita e Leitura: Implementadas na classe KeyValueStoreGrained, as operações put e get permitem que pares chave-valor sejam armazenados e recuperados de forma eficiente. A granularidade dos locks garante que múltiplas operações em chaves diferentes possam ocorrer simultaneamente sem interferência.

Operações Compostas: As operações multiPut e multiGet foram projetadas para manipular múltiplos pares chave-valor de maneira atômica. A implementação utiliza ordenação de chaves para evitar deadlocks e assegura que todos os pares sejam processados de forma consistente, mesmo sob alta concorrência.

Leitura Condicional: A funcionalidade getWhen permite que uma chave seja recuperada apenas quando uma condição específica em outra chave seja atendida. Para isso, são utilizadas condições associadas a locks, permitindo que as threads esperem eficientemente até que a condição seja satisfeita.

Gerenciamento de Conexões Concorrentes: O SessionManager foi projetado para limitar o número de conexões simultâneas ao servidor. Essa limitação é implementada utilizando locks e condições, garantindo que conexões excedentes aguardem até que recursos sejam libertados.

3.2. Estrutura do Código

A estrutura do código é modular e bem organizada, dividida em pacotes que separam claramente as responsabilidades.

Tem-se o pacote do server, que inclui as classes principais para gerenciar conexões (ServerConnector), autenticação (AuthManager), sessões (SessionManager) e o processamento das solicitações dos clientes (ServerMain).

O pacote do client que contém a API cliente (ClientAPI), que encapsula a lógica de comunicação com o servidor, e a interface de linha de comando (ClientInterface), usada para interação direta com o útilizador.

O pacote common fornece componentes reutilizáveis, como o armazenamento de dados (KeyValueStore-Grained), a fila de requisições (RequestQueue) e o pool de threads (ThreadPool), além de utilitários para sincronização e concorrência.

E finalmente, o pacote protocol em que define o protocolo de comunicação binária, com classes como TaggedConnection e TaggedFrame, que gerenciam a serialização e desserialização de mensagens, além de garantir a integridade dos dados transmitidos.

3.3. Decisões de Projeto

Algumas decisões de projeto foram críticas para o sucesso do sistema:

Uso de Locks Granulares: Optou-se por implementar locks por chave no armazenamento (KeyValueStore-Grained) para minimizar contenções. Essa abordagem permite que múltiplas threads realizem operações simultaneamente em diferentes chaves sem interferir umas nas outras.

Protocolos Etiquetados: A introdução de etiquetas nos quadros de comunicação (TaggedFrame) simplificou o processamento de mensagens no servidor e no cliente assim como a identificação e o roteamento das operações.

Thread Pool Centralizado: A utilização de um pool de threads no servidor garantiu que o número de threads ativas fosse controlado e preveniu sobrecarga de recursos.

Atomicidade nas Operações: Todas as operações compostas foram projetadas para serem atómicas para garantir a consistência no armazenamento.

Com essa organização, o sistema está preparado para suportar cargas de trabalho variadas e fornecer uma base sólida para futuras extensões e melhorias.

Todas as funcionalidades e requisitos foram implementados desde as funcionalidades básicas às funcionalidades avançadas. Os requisitos também foram respeitados e tidos sempre em mente na realização do projeto de forma a respeitar as normas.

4. Testes e resultados

4.1. Cenários de Teste

Os testes foram realizados com diferentes configurações de workload para avaliar o comportamento do sistema em diversos cenários. Cada teste foi projetado para medir métricas como throughput e escalabilidade. Para cada workload foram simulados 6 testes, tendo cada uma duração igual a dez segundos. Para a interpretação dos gráficos os "Nops" representam o número de operações realizadas no intervalo de tempo.

Os principais workloads que foram realizam-se dividem-se em:
- heavy read: taxa de escrita de 10%.
- heavy write: taxa de escrita de 90%.
- balanced: taxa de escrita de 50%.
- big set: 5000 chaves
- small set: 1000 chaves
- high connection: maior número de clientes conectados assim como o número de threads concurrentes em cada um a enviar pedidos.
- low connection: menor número de clientes e menor número de threads por cliente.

Utilizamos uma Classe Benchmark que foi responsável por gerar e atribuir a cada cliente um conjunto de operações a enviar ao servidor para que no final fosse possível medir o desempenho. Os workloads foram organizados em ficheiros separados e geramos um script para que o Benchmark fosse capaz de ler o ficheiro *properties* e realiza-se o teste pedido.

4.1.1. Métricas Avaliadas

Com base nas configurações de workload que nós criamos, tivemos em conta principalmente o número de operações que estipulamos num determinado tempo, e por consequência o Throuput (operações por segundo).

No contexto dos testes realizados, a distribuição de workloads de tipo uniforme e zipfian desempenharam papéis importantes no desenvolvimento do projeto, pois permitiram avaliar diferentes cenários de uso e verificaram a robustez e a eficiência em condições distintas:

- Distribuição Uniforme: Ajudou a simular situações em que o acesso aos dados era igualmente distribuído entre todas as chaves, validou a eficiência do thread pool e do uso de locks granulares ao demonstrar que o sistema podia lidar com cargas homogéneas de maneira eficaz e foi essencial para testar a escalabilidade do sistema sem criar áreas de alta contenção, permitindo identificar gargalos gerais na arquitetura do servidor.
- Distribuição Zipfian: Refletiu padrões de acesso típicos do mundo real, onde algumas chaves são significativamente mais acedidas do que outras, expôs as limitações do sistema sob cargas desequilibradas, destacando a necessidade de otimizações como balanceamento de carga e estratégias para mitigar áreas de alta contenção e foi útil para avaliar o comportamento do sistema em cenários de alta contenção.

4.2. Resultados

4.2.1. Teste em Big Set com número de threads variável

Neste primeiro conjunto de testes focamo-nos em verificar a escalabilidade de cada workload alteranto o número de threads (ServerThreads, ClientThreads) que correspondem ao número de conexões de clientes e ao número de threads por cliente a submeter pedidos. Utilizamos testes de 10 segundos neste relatório de forma a perceber as diferenças à medida que alteravamos parámetros nas workloads.

Especificações do Computador de Testes

- Sistema Operativo: Pop!OS 22.04 LTS (x86_64)
- Processador (CPU): Intel Core i7-12700H (12ª geração, 20 núcleos, até 4.6 GHz)
- Placa Gráfica (GPU): Mesa Intel® Graphics (ADL GT2)

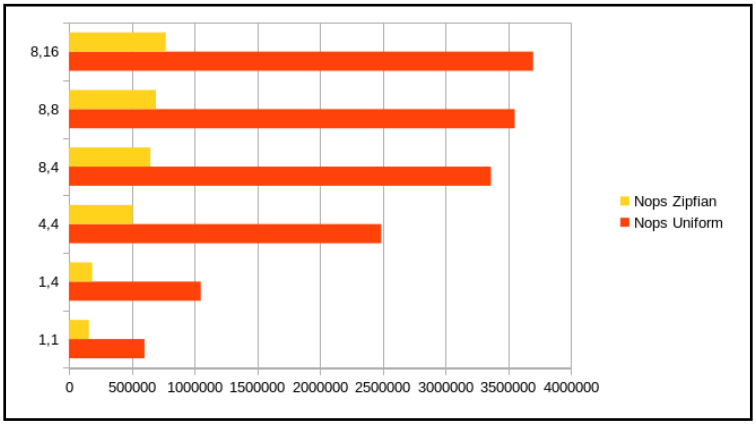

Figura 2: Read Heavy Workload Results

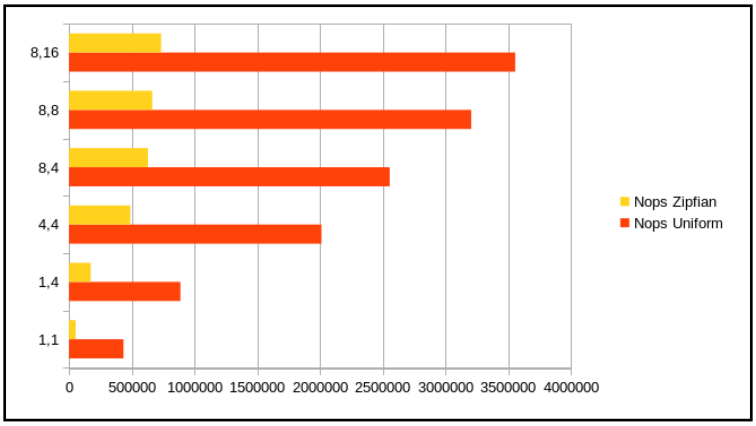

Figura 3: Write Heavy Workload Results

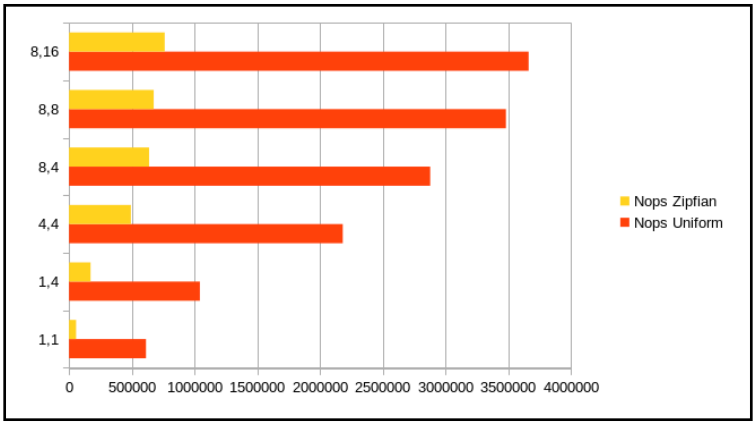

Figura 4: Balanced Workload Results

Primeiramente, e como esperado, é vísivel a escalabilidade do sistema à medida que aumentamos o número de threads como deveria ocorrer de modo natural.

A partir dos gráficos anteriores, é notória uma diferença significativa na contenção da nossa arquitetura quando aplicada a distribuição uniforme e a distribuição zipfian, uma vez que devido à natureza enviesada do zipfian, poucas chaves recebem a maior parte dos acessos, criando "pontos quentes" de contenção. Em contraste, a distribuição uniforme, faz com que os acessos entre as chaves sejam equiprováveis, resultando assim numa menor contenção.

Uma característica menos percetível do nosso design do sistema é a diferença do número de operações entre o heavy write e heavy read. Apesar que o heavy read tenha obtido melhores resultados no geral, é evidente que tanto a maneira de como o ThreadPool trabalha como o modelo de KeyValueStore que escolhemos (Lock por chave) contribuem para que, mesmo que existam acessos em pontos quentes, é ainda raro ocorrer uma contenção tão significativa numa chave que cause impactos relevantes.

Quanto ao workload balanced, verificou-se que o seu desenpenho fica no centro dos workloads read heavy e write heavy assim como esperado, dado que a sua taxa de escrita está definida a 50% esperamos um desempenho alinhado na metade do desempenho dos outros dois workloads referidos.

4.2.2. Teste em Small Set com número de threads variável

No segundo conjunto de testes também focamo-nos em verificar a escalabilidade de cada workload alteranto o número de threads (ServerThreads, ClientThreads) que correspondem ao número de conexões de clientes e ao número de threads por cliente a submeter pedidos.

Especificações do Computador de Testes
- Sistema Operativo: Ubuntu 24.04.1 LTS (x86_64)
- Processador (CPU): AMD Ryzen 5 5500U (6 núcleos com 2 threads cada, até 4GHz)
- Placa Gráfica (GPU): Radeon Graphics

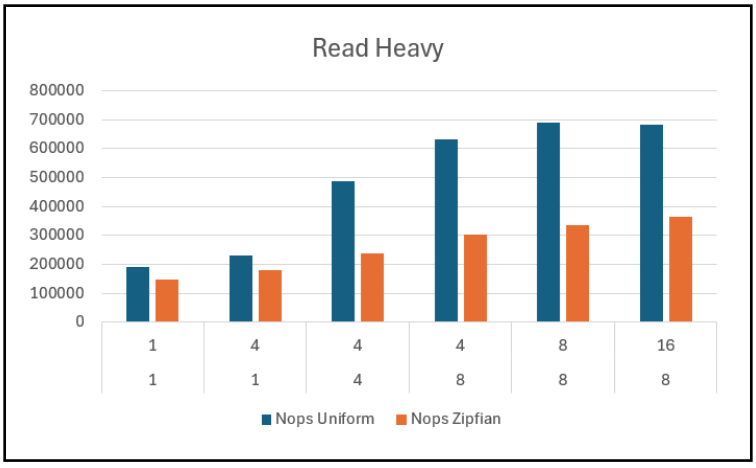

Figura 5: Read Heavy Workload Results

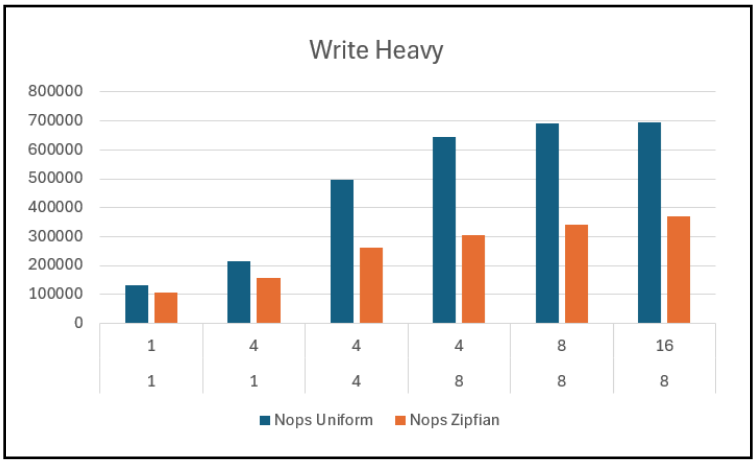

Figura 6: Write Heavy Workload Results

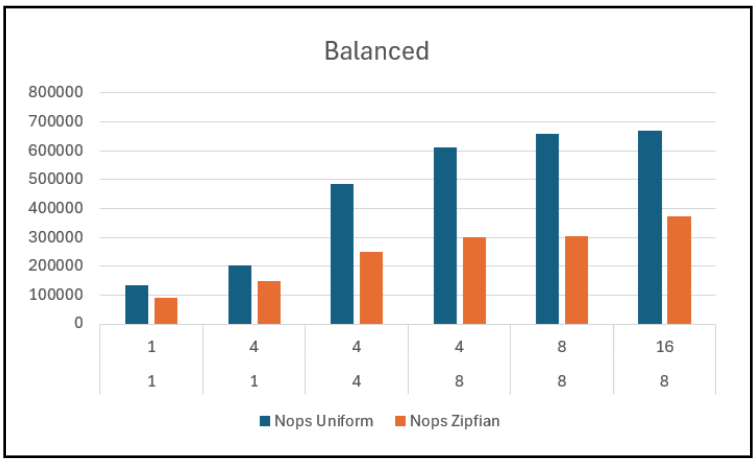

Figura 7: Balanced Workload Results

Primeiramente, verifica-se que com o aumento de threads o número de operações também aumenta demonstrando a escabilidade esperada.

Em relação ao primeiro teste, o comportamento foi semelhante e decidimos incluir outra máquina de testes para perceber o desempenho em diferentes componentes de hardware.

Através dos gráficos, podemos verificar ainda que existe uma proporção estimada na metade do desempenho da distribuição uniforme com a zipfian, muito maior do que a proporção em Big Set.

4.2.3. Comparações finais

A partir dos dois tipos de testes executados, ainda que com máquinas diferentes, a proporção entre as distribuições diz-nos mais sobre o nosso sistema no que toca ao número de chaves presentes na KeyValueStore. Percebemos que quanto maior for o nosso 'Set' de chaves, menos desempenho iremos obter por parte do servidor, e isto deve-se ao facto de termos definido a nossa 'store' como sendo 'lock per key'. Quanto mais chaves possuir o servidor na sua 'store', mais memória e mais controlo terá de ser utilizado para organizar os bloqueios de cada chave individualmente. Esse aumento no número de chaves resulta em um maior overhead de bloqueios, o que leva a uma maior latência nas operações de leitura e escrita, além de uma maior contenção entre threads ou processos concorrentes, causanto um impacto negativo no desempenho do sistema.

Estamos conscientes de que outras estratégias poderiam ter sido utilizadas, nomeadamente sharding para melhorar o desempenho deste requisito, no entanto iria afetar outros aspetos, trantando-se assim de um trade-off que teria de ser decidido por nós.

5. Conclusões e Trabalho Futuro

Concluímos que o sistema desenvolvido é capaz de suportar as funcionalidades básicas e avançadas exigidas e ainda fornece um armazenamento chave-valor distribuído eficiente e escalável. O uso de sincronização fina e técnicas avançadas de concorrência demonstrou-se eficaz na manutenção de desempenho. As estratégias adotadas mostraram-se eficiente em termos de escalabildade e contenção, contudo quanto maior for a quantidade de chaves armazenadas, a latência de operações é comprometida bem como o overhead de bloqueios. O projeto foi certamente uma extensão dos conhecimentos adquiridos ao longo do semestre na unidade curricular de Sistemas Distribuídos.