Universidade do Minho
Licenciatura em Engenharia Informática

Sistemas Distribuídos

Trabalho Prático Cloud Computing

Grupo 14

**<< João Magalhães (A100740), Jorge Rodrigues (A101758),
Rodrigo Gomes (100555), Miguel Pereira (A91971) >>**

Ano Letivo de 2023/2024

Índice

Introdução

O presente relatório aborda a implementação de um serviço de cloud computing, focalizado na funcionalidade "*Function-as-a-Service*" (FaaS). Este projeto visa proporcionar aos utilizadores a capacidade de enviar tarefas de computação a partir de um cliente, sendo estas executadas num servidor assim que houver disponibilidade. Uma característica distintiva deste serviço é a consideração primordial da memória como fator limitante nos servidores. Esta é contabilizada através da soma das memórias individuais dos *workers* registados.

A natureza do código das tarefas a executar, assim como os respetivos resultados, é simplificada para um *array* de bytes (byte[]), proporcionando eficiência e agilidade no processamento. O cerne do serviço reside na habilidade de manter uma fila de espera eficiente de tarefas a serem executadas, garantindo simultaneamente uma otimização do uso dos recursos disponíveis, especialmente no que concerne à gestão da memória no servidor.

Para assegurar uma execução eficaz e evitar congestionamentos, é essencial implementar estratégias que previnam a sobrecarga de pedidos concorrentes, garantindo que estes não excedam a capacidade máxima de memória disponível. Adicionalmente, o serviço deve evitar que alguns pedidos fiquem perpetuamente em espera, enquanto outros são continuamente priorizados.

Este relatório detalhará a arquitetura, implementação e estratégias adotadas para alcançar os objetivos propostos apresentando as soluções adotadas para garantir a eficiência e a otimização na execução das tarefas.

Arquitetura do projeto

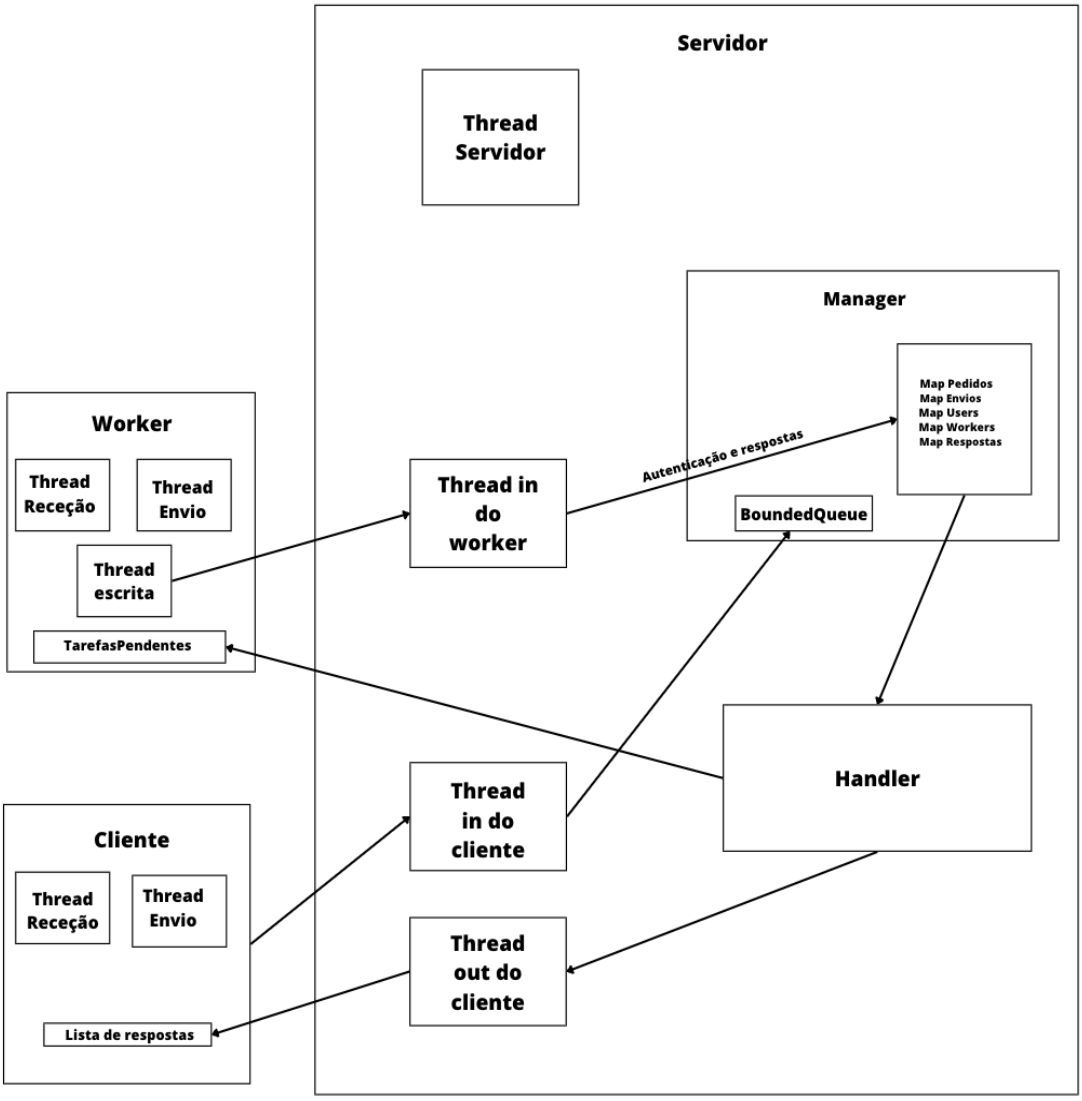

Figura 1 - Arquitetura do serviço

Estruturas importantes

Classe Mensagem

É imperativo ressaltar a estrutura da classe "Mensagem", uma vez que desempenha um papel central e extensivo ao longo de todo o projeto. Esta classe é essencial para a comunicação entre componentes, pois encapsula informações cruciais para a execução e gestão eficiente do serviço.

Cada instância da classe "Mensagem" é composta pelos seguintes elementos:

1. **Código de Identificação:** Um identificador único que permite distinguir cada mensagem, facilitando o roteamento e processamento eficaz.

2. **Tipo da Mensagem:** Categoriza a mensagem em diferentes tipos, como Resposta, Tarefa, Autenticação, entre outros. Essa classificação é fundamental para a correta interpretação e tratamento da mensagem no contexto do serviço.
3. **Credenciais do Remetente:** Incluindo o nome e a senha do remetente, garantindo a autenticidade e segurança na troca de informações.
4. ***Worker* Responsável (se aplicável):** Identifica o *worker* responsável pelo envio da mensagem de resposta. Esta informação é essencial para facilitar a atualização da memória do *worker* associado.
5. **Inteiro Auxiliar:** Um parâmetro dinâmico que desempenha diferentes papéis em casos específicos. Pode indicar a memória total do *worker* ou o número de tarefas pendentes no servidor, fornecendo informações adicionais necessárias para o processamento adequado.
6. ***Array* de Bytes:** Este componente armazena os bytes essenciais da mensagem a ser transmitida. É crucial para a transmissão eficiente e correta das informações associadas à mensagem.
7. **Caminho do Arquivo (se aplicável):** Aquando da leitura de tarefas do cliente proveniente de um ficheiro, este campo contém o caminho desse arquivo a ser lido, facilitando a obtenção de dados necessários para a execução de tarefas específicas.

Classe Wrapper

Identificamos a necessidade premente de criar a classe "Wrapper", uma entidade fundamental para facilitar a leitura de mensagens com a *tag* "Resposta". O propósito primordial desta classe é otimizar o processo de gerir as respostas destinadas a clientes específicos. Para atingir este objetivo, o "*Wrapper*" é utilizado numa estrutura de dados do tipo map (*key* = nome do cliente, *value* = *Wrapper*).

A estrutura do "Wrapper" compreende os seguintes elementos essenciais:

1. **Variável de Condição Específica:** Uma variável de condição particular, habilitando o "adormecer" eficiente da *thread* associada ao cliente quando a lista de pedidos (pb) está vazia. Da mesma forma, essa variável permite o despertar imediato quando a lista de pedidos de resposta não está vazia.
2. **Lista de Pedidos (pb):** Mantém um registo das mensagens de resposta pendentes a serem enviadas ao cliente. Esta lista é crucial para garantir que todas as respostas sejam entregues ao cliente na ordem apropriada.

É relevante destacar que todas as operações relacionadas às variáveis de condição dependem de um único *Lock*. Além disso, a instância da classe "Wrapper" é armazenada durante a autenticação do cliente, proporcionando uma associação direta e eficaz entre a *thread* do cliente e as respostas esperadas.

Classe WorkerWrapper e Conta

Precisamos desenvolver mais duas classes: "WorkerWrapper" e "Conta". A primeira é composta por uma instância *worker* juntamente com uma lista de pedidos em execução pelo *worker*. Esta abordagem eficiente permite nos determinar a memória disponível desse *worker* de forma precisa.

Por sua vez, a classe "Conta" também é de grande importância, pois não só nos possibilita bloquear o cliente associado, devido ao *lock* que está contido na classe, evitando assim o bloqueio total da estrutura de clientes, mas também inclui uma lista de pedidos de tarefa provenientes do cliente.

Classe Cliente

Por fim é necessário destacar duas classes cruciais no projeto: a classe "Cliente" e a classe "Worker". Na classe "Cliente", encontramos um componente vital denominado *Map<String, List<String>>* respostas, implementado como um *HashMap*. Este mapa é utilizado para armazenar as respostas previamente obtidas pelo cliente, facilitando consultas futuras. O diferencial desta classe reside nas suas duas *threads* essenciais: uma dedicada à receção de dados e outra à transmissão. Este design permite que o cliente mantenha a sua operação ininterrupta, mesmo quando operações simultâneas são realizadas.

Classe Worker

No caso da classe "Worker", observamos um conjunto similar de elementos fundamentais. O componente central é uma *boundedQueue* denominada tarefasPendentes, que regista as tarefas que o *worker* precisa executar. Além disso, um *reentrantLock* (l) é empregue para "adormecer" o *worker* quando não há tarefas pendentes. A condição "espera" é crucial para coordenar o *Lock*, utilizando *await* e *signal*. Têm ainda um atributo memoriaTotal para acompanhar a memória total do *worker*, memoriaUsada para monitorizar a quantidade de memória em utilização, e informações de autenticação como nome e senha. Assim, semelhante à classe "Cliente", a classe "Worker" incorpora duas *threads* essenciais: uma para receber as tarefas e outra para enviar as respostas correspondentes. Adicionalmente a classe "Worker" necessita de mais uma *thread* que escreve as respostas a serem enviadas ao servidor evitando assim misturas de respostas quando o *worker* executa mais do que uma tarefa simultaneamente.

Essa abordagem de design, com foco em *threads* distintas para operações de receção e envio e escrita, é crucial para manter a eficiência e a responsividade do sistema, garantindo uma execução suave e simultânea das operações tanto para clientes quanto para *workers*.

Estratégia Adotada

A estratégia adotada neste projeto incorpora abordagens distintas para o tratamento de diferentes tipos de pedidos, otimizando a eficiência e a gestão de recursos. Em primeiro lugar, destacamos a gestão dos pedidos de autenticação, os quais são tratados de imediato sem passar pela *thread handler,* Esta decisão visa evitar congestionamentos que poderiam surgir se esses pedidos fossem encaminhados para o buffer (*boundedqueue*). O buffer, implementado como uma *priorityQueue*, que ajuda a formular uma ordem já preferível para quando os pedidos são retirados desta estrutura.

No que diz respeito às tarefas enviadas pelos clientes que requerem um *worker*, esses pedidos são adicionados ao buffer. O buffer é esvaziado sempre que a *handler* termina o

escalonamento dos pedidos anteriores, sendo então, atribuídos aos *worker*s registados que, no momento, apresentam as melhores condições para a execução da tarefa, considerando a disponibilidade de memória (tentamos atribuir sempre ao *worker* com menos memória disponível capaz de as receber, começando por pedidos maiores). Caso não haja *worker*s disponíveis, uma mensagem de resposta é criada automaticamente para o pedido enviado. No caso de haver *workers* , mas nenhum estiver apto a executar a tarefa, o serviço aguarda até que as condições sejam satisfeitas, evitando assim a questão da "starvation" dos pedidos.

Os pedidos de consulta por parte dos clientes, que não exigem *workers*, são tratados de maneira semelhante às tarefas que os requerem, sendo colocados no buffer. No entanto, a informação que se deseja obter com esses pedidos é obtida diretamente na classe "Manager". Esta classe abrange uma ampla gama de informações do serviço, incluindo clientes e *worker*s registados, respostas dos clientes, pedidos pendentes e pedidos a serem enviados aos clientes.

Um componente crucial desta estratégia é a resposta garantida para cada pedido enviado pelo cliente. Todas as respostas, envio pendentes e pedidos dos clientes provenientes são armazenadas em estruturas de dados "Map", com o auxílio de duas classes criadas por nós denominada *Wrapper* e Conta. Uma *thread* no servidor, criada quando a autenticação do cliente é concluída, permanece adormecida enquanto o *Wrapper* associado ao cliente na "Map" esteja vazio, assegurando um fluxo ordenado e eficiente de respostas aos clientes. Esta abordagem multifacetada reflete a dedicação em garantir uma gestão eficiente e equitativa dos recursos disponíveis, resultando em um serviço de cloud computing robusto e responsivo.

Em termos de otimização tivemos em conta o uso de *ReentrantReadWriteLock* na classe "Manager" dado que é constantemente consultada para leituras permitindo que várias *threads* possam aceder às suas informações, no entanto, apenas uma delas possa escrever de cada vez na classe (as escritas no "Map" são pouco frequentes). Esta implementação torna o serviço mais flexível aprimorando a responsividade do sistema proporcionando um equilíbrio entre concorrência e exclusão mútua.

Reflexão crítica e aprendizagem

Nesta secção, vamos abordar algumas decisões de design arquitetural do serviço, assim como a solução implementada pelo grupo e algumas considerações.

Primeiramente, o escalonamento. Inicialmente, a primeira solução implicava na mesma o uso de uma fila de espera com prioridade (com a devida concorrência tratada) onde o pedido era retirado sequencialmente. O problema desta implementação é que, mesmo com o aumento da prioridade dos pedidos, poderia levar a alguma "starvation". Além disso, ainda estávamos a escolher os pedidos mais pequenos e a colocar no *worker* com mais memória disponível capaz de os executar, o que não é muito eficiente em termos de memória. Assim, avançamos para a estratégia atual que visa combater estes dois pontos. Ao retirar múltiplos pedidos de cada vez, deixamos de ter starvation, pois a *handler* só procura novos pedidos depois de todos os retirados forem executados. No que toca à atribuição, utilizamos algo semelhante ao algoritmo "Best fit", onde atribuímos ao *worker* com memória disponível suficiente mais pequena para o pedido. Contudo esta situação não é perfeita, pois à aspetos como o total de

pedidos retirados de cada vez que pode ser ajustado, e ainda dependendo do tamanho dos pedidos, outras estratégias mais otimizadas poderiam ser mais apropriadas. Além disso, com o aumento dos *workers* conseguimos manter um uso eficiente da memória, mas o custo computacional também aumenta.

Segundo ponto é a concorrência dentro do servidor. De modo geral, o grupo está satisfeito com a performance do servidor e, de modo geral, achamos que as secções críticas estão bem protegidas e granuladas em casos onde se justifica (auxiliadas pelas classes como Conta ou Wrapper). Contudo, devido à baixa afluência de *threads* diferentes que interagem com uma estrutura particular, nesse caso usamos apenas um “lock” e uma “condição”. Esse caso é precisamente a estrutura que guarda informações sobre os *workers* onde, exceto a autenticação de cada worker, que é realizada pela *thread* gerada na conexão, apenas a handler opera nessa região, daí não ter existido grande necessidade de granulação. Contudo, se a aplicação for esperada num cenário onde existem imensas autenticações de *workers*, possivelmente compensaria optar por uma estratégia mais granular.

Por fim, resta apenas falar das mensagens. De modo geral, achamos que estão eficientes, sendo que cumprem perfeitamente o propósito. Contudo, sentimos que ao utilizar algo fixo acabamos por passar muitos “bytes” desnecessários, onde talvez poderíamos optar antes por um “cabeçalho”, onde diminuiríamos o “overhead” em troco de uma desrealização mais complexa.

Conclusão

Em suma, o desenvolvimento e implementação do serviço de cloud computing com funcionalidade "Function-as-a-Service" (FaaS) revelou-se um desafio que culminou numa solução robusta e eficiente. O cerne do projeto reside na capacidade de processar tarefas de computação de forma ágil, considerando a memória como fator limitante no servidor.

A introdução da classe "Mensagem" como elemento central para a comunicação entre componentes proporcionou uma estrutura organizada e eficaz. O uso de *arrays* de bytes simplificou a representação do código das tarefas, contribuindo para a eficiência na transmissão e execução.

A criação da classe "Wrapper" mostrou-se indispensável para a gestão de respostas destinadas aos clientes, oferecendo uma solução elegante e competente. A estratégia adotada, que inclui a priorização de pedidos, a gestão de tarefas com e sem *workers*, reflete a dedicação em otimizar a utilização dos recursos disponíveis.

A implementação de buffers e a utilização de estruturas como filas de prioridade contribuíram para a organização e gestão eficaz das tarefas, evitando congestionamentos e garantindo uma execução fluida do serviço.

Em síntese, o serviço de cloud computing desenvolvido apresenta-se como uma solução sólida, equilibrando eficiência e gestão otimizada de recursos. As estruturas implementadas e a estratégia adotada visam atender às demandas dos utilizadores, garantindo uma resposta eficaz e um serviço responsivo.