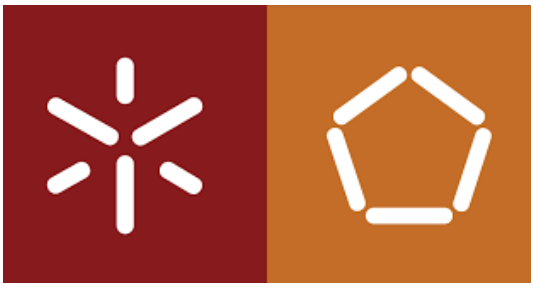

Universidade do Minho

Licenciatura em Engenharia Informática

Sistemas Distribuídos

Trabalho Prático - Grupo 18

Diogo Marques - A100897

Roberto Capela - A100655

Lingyun Zhu - A100820

Rui Alves - A100699

30 de dezembro de 2023

Índice

Introdução

No âmbito da Unidade Curricular de Sistemas Distribuídos foi-nos apresentado um trabalho prático que visava a comunicação entre um servidor e vários clientes, de modo a que fossem executadas determinadas tarefas e apresentados alguns dados estatísticos sobre o estado geral do sistema.

Numa fase inicial pensámos em desenvolver o projeto de forma gradual, ou seja, começar por assegurar as funcionalidades básicas e só depois passar às mais avançadas, contudo reparámos que isso implicaria refazer parte da implementação do servidor, como tal abandonámos essa ideia e definimos de imediato uma arquitetura para um sistema com várias máquinas a executarem tarefas.

Posto isto, e tendo em consideração a abertura do enunciado do trabalho prático, pretendemos apresentar todas as decisões que tomámos, bem como as estratégias que adotámos a fim de solucionar os problemas com os quais nos fomos deparando.

Arquitetura

Durante as aulas teóricas foi apresentada uma arquitetura cliente-servidor na qual as estruturas utilizadas dos dois lados eram bastante semelhantes, dando a sensação que certas partes do código eram partilhadas, como tal procurámos montar uma arquitetura desse género, visto que obteríamos uma implementação significativamente mais genérica.

Cliente

Um dos principais problemas na arquitetura do cliente, reside no facto do servidor poder bloquear ao enviar pacotes de resposta (blocos de informação), pois se o *buffer* do *socket* nunca for despejado, este eventualmente acabará por ficar cheio e será impossível que o servidor continue a enviar dados.

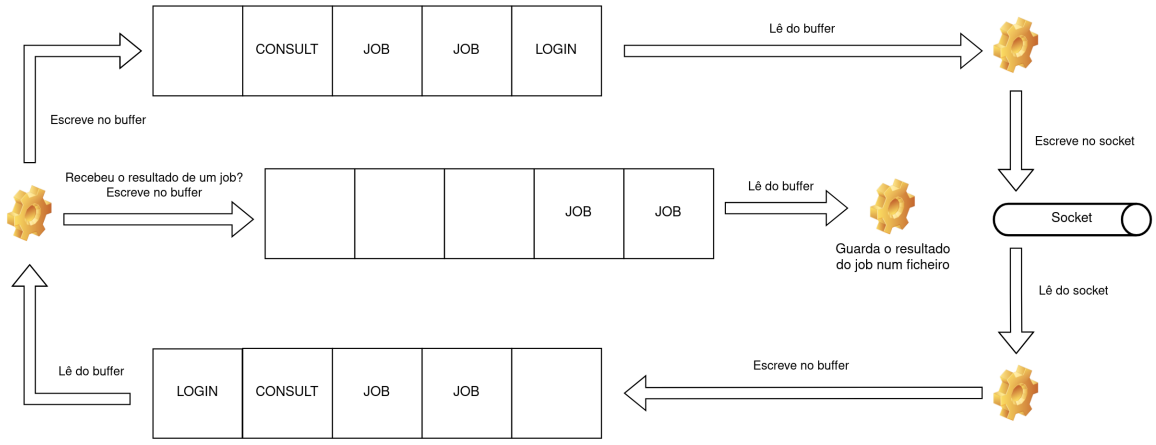

Figura 1. Arquitetura do cliente

Thread Principal

É responsável por interagir com o utilizador e criar os pedidos conforme as informações que este fornece, por sua vez os pedidos são encapsulados em pacotes e enviados para um *buffer de saída*. Quando o utilizador solicita a recolha de pacotes, é efetuada uma leitura do *buffer de entrada*, e caso algum dos pacotes recebidos seja referente a um *job*, o mesmo é encaminhado para um *buffer de escrita*.

Threads do Socket

Existem duas *threads* com acesso ao *socket*, uma delas lê os pacotes criados pela *thread principal* e escreve-os no *socket*, enquanto que a outra está constantemente a ler pacotes e a escreve-los no *buffer de entrada*, evitando assim que o servidor bloqueie ao enviar dados.

Thread de Escrita

A escrita dos resultados de um *job* é uma tarefa que requer algum esforço computacional, daí termos optado por criar uma *thread* e um *buffer* auxiliar. Neste caso, a *thread principal* coloca apenas *jobs* bem sucedidos no *buffer* em questão, como tal é possível obter imediatamente a informação que se pretende guardar em disco sem verificar o tipo do pacote.

Problemas Arquiteturais

Ao analisarmos esta arquitetura identificamos um erro que resulta de uma má prática do utilizador, se o mesmo estiver constantemente a enviar pedidos sem recolher as respetivas respostas, o *buffer de entrada* acabará por ficar saturado.

Numa eventual solução, *thread principal* poderia efetuar leituras periódicas do *buffer*, todavia isso levaria à apresentação de respostas no *stdout* sem a autorização do utilizador.

Máquina

O esquema arquitetural de uma máquina é bastante semelhante ao do cliente, o que permitiu reutilizar algum do código desenvolvido, em especial aquele que é referente aos *buffers*.

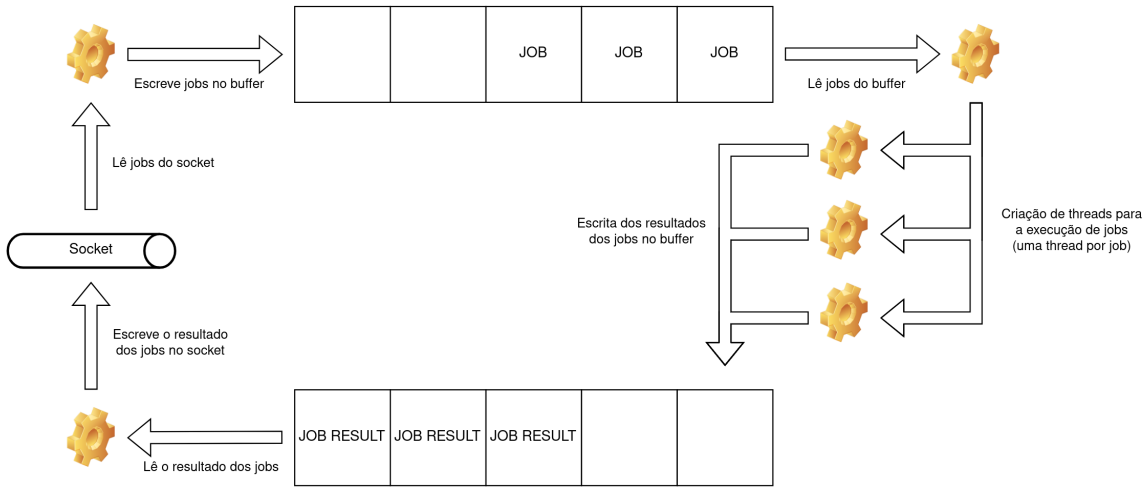

Figura 2. Arquitetura da máquina

Thread Principal

Assim que um *job* é atribuído a uma máquina, o mesmo é encapsulado num pacote e enviado pelo servidor através de um *socket*, sendo que de seguida será recolhido e colocado num *buffer de entrada*. Posto isto, a *thread principal* tem como função ler os pacotes contidos nesse *buffer*, e lançar as *threads* necessárias à execução dos próprios *jobs*.

Thread de Execução

Cada tarefa é executada por uma *thread*, sendo que no final o resultado é encapsulado num pacote e inserido num *buffer de saída*. A princípio julgámos que seria mais vantajoso criar uma *thread pool*, visto que isso permite a reutilização de *threads*, contudo isso implicaria a utilização de classes e métodos que não foram abordados nas aulas.

Servidor

O servidor é claramente a entidade com a arquitetura mais complexa, pois por um lado tem de apresentar estruturas de dados com controlo de concorrência, e por outro necessita de fazer uma clara distinção entre os tipos de pacotes que recebe, já para não mencionar que é primordial um bom escalonamento dos *jobs*.

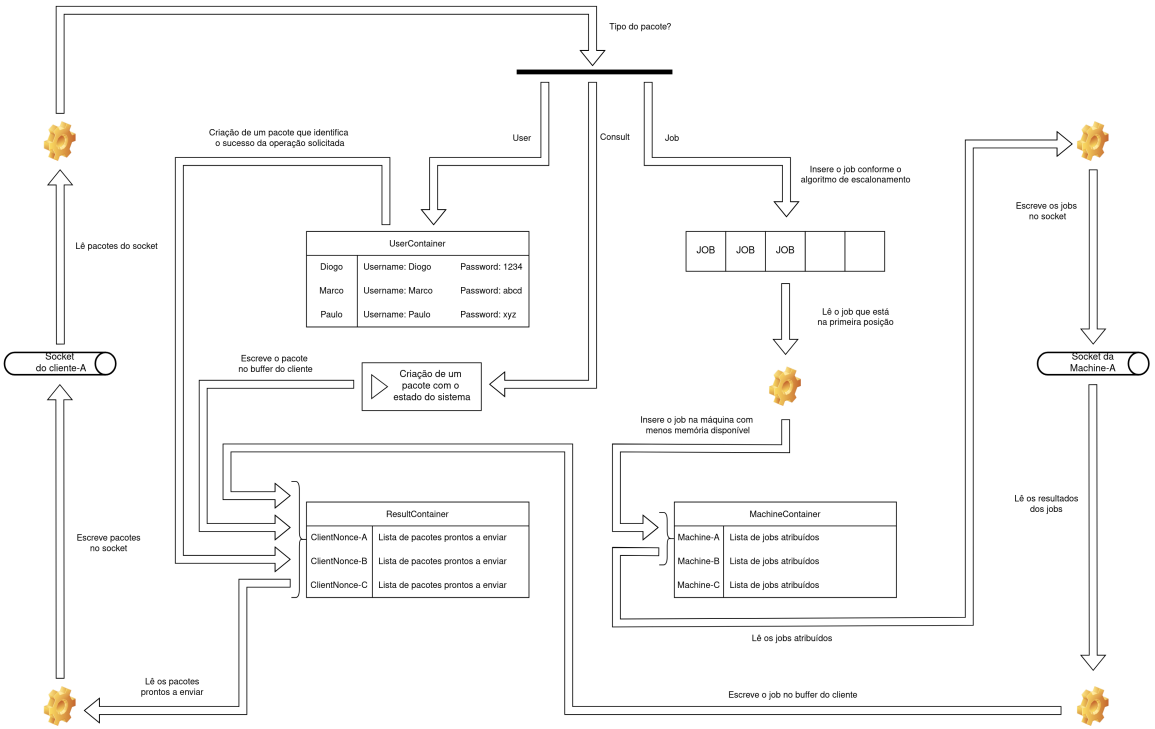

Figura 3. Arquitetura do servidor

UserContainer

Antes de um cliente poder usufruir dos serviços disponibilizados pelo sistema, o mesmo deve criar uma conta ou efetuar *login*, como tal é enviado um pacote *User* através do *socket*, pacote esse que é imediatamente recebido pelo servidor.

Conforme o conteúdo do pacote e o seu protocolo, é feita uma análise do *Map* de utilizadores, e criado um novo pacote que indica se a operação requerida foi bem sucedida, de notar que esse pacote será inserido no *ResultContainer*.

No fundo esta estrutura de dados é essencialmente um *Map* de utilizadores, que contribui para assegurar as funcionalidades de *login* e criação de conta.

MachineContainer

No momento em que uma máquina se conecta ao servidor, é realizada a atribuição de um *nonce*, que passa a funcionar como seu identificador, dessa forma, quando o servidor pretende enviar um *job* para uma determinada máquina, basta utilizar o seu *nonce* para selecionar a entrada correta do *Map*, e assim adicionar o *job* à lista de tarefas que a máquina deverá executar.

Depois do *job* ser atribuído, as diversas *threads* de leitura serão acordadas e uma delas passará o pedido à máquina em questão através do respetivo *socket*.

ResultContainer

A exemplo do caso anterior, quando um cliente se conecta ao servidor, é-lhe atribuído um *nonce* que passa a servir de identificador, desta forma é possível evitar que um cliente recolha respostas que não lhe foram dirigidas.

A princípio pensámos em empregar o *username* do utilizador como chave do *Map*, contudo isso impossibilitava que um utilizador tivesses múltiplas conexões em paralelo, visto que dessa forma haveriam várias *threads* a tentar ler da mesma entrada.

Locks

Todas as estruturas de dados usufruem de um *ReentrantLock*, à exceção do *UserContainer* que apresenta um *ReentrantReadWriteLock*, visto que as consultas são significativamente mais frequentes que as escritas.

Escalonamento de Tarefas

Quando abordámos o problema da gestão dos *jobs*, a solução que nos veio imediatamente à cabeça visava a utilização de um *fifo*, todavia percebemos rapidamente que existiam várias limitações, sendo a mais flagrante o facto de um *job* grande bloquear a execução de outros mais pequenos.

Tendo isto em mente, definimos duas métricas que em conjunto indicam a prioridade de um determinado *job*.

- **Tolerância:** Indica o número de vezes que um *job* pode ser ultrapassado na fila de espera do servidor, por outras palavras, um *job* com tolerância zero nunca será ultrapassado.
- **Memória:** Funciona como elemento de comparação entre *jobs*, ou seja, um determinado *job* apenas será ultrapassado por outro com menos memória.

Estando a prioridade de um *job* perfeitamente definida, podemos elaborar uma variante do *bubble sort* que tem em consideração estes dois fatores, tolerância e memória.

1. O *job* é inserido na última posição do *buffer*.
2. Se o *job* anterior possui mais memória que o atual, e a sua tolerância é diferente de zero, fazemos *swap* e decrementamos a tolerância em uma unidade.

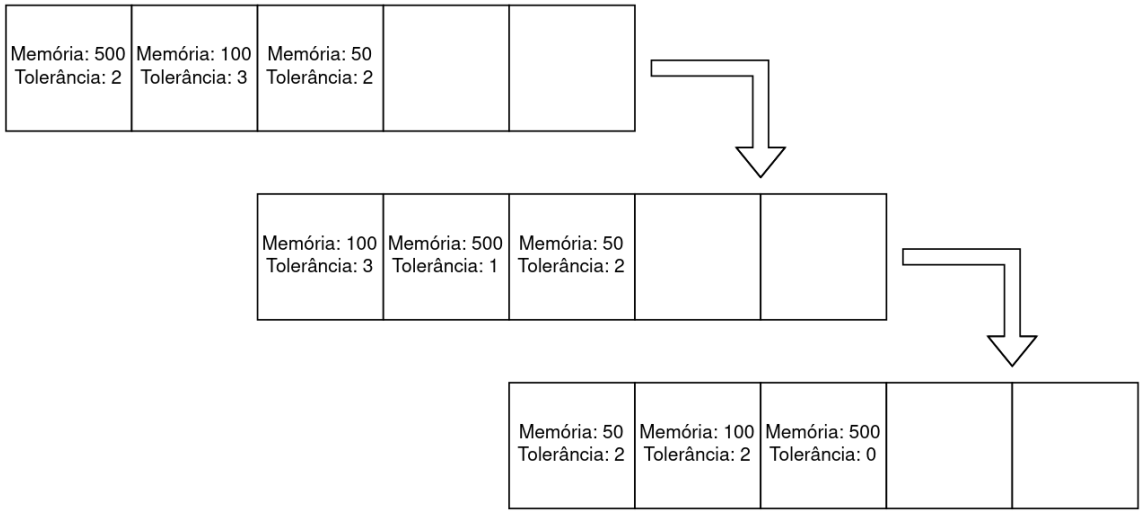

Figura 4. Escalonamento de três *jobs*

Tal como podemos constatar pelo exemplo, o *job* de memória 500 foi ultrapassado duas vezes e colocado no fim, consequentemente a sua tolerância foi decrementada para zero.

Já em relação ao *job* de memória 50, o mesmo era considerado inferior aos restantes, como tal passou para o início sem que a sua tolerância fosse alterada.

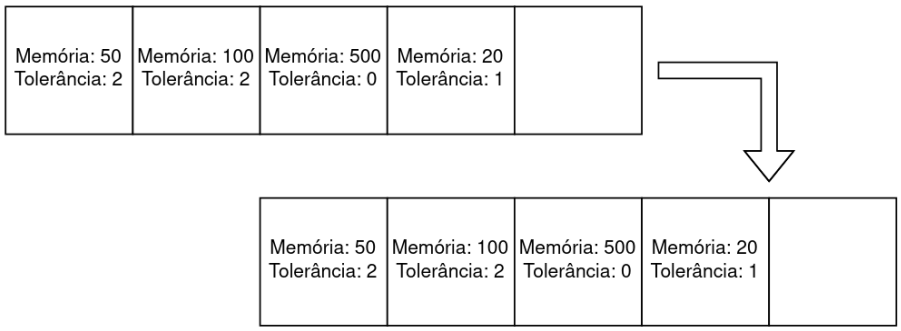

Figura 5. Escalonamento de quatro *jobs*

Nesta situação foi adicionado um *job* que na prática é inferior aos restantes, contudo o *job* que o precede tem tolerância zero, assim sendo não é realizado qualquer *swap*, e o algoritmo termina porque encontrou um *job* mais prioritário que o atual.

Tendo esta noção de prioridade, é possível evitar que os *jobs* mais longos bloqueiem a fila de espera, sem que para isso sofram de *starvation*, visto que uma tolerância nula impossibilita ultrapassagens.

Depois de estar definido o *job* que dever ser executado (elemento de índice zero), o mesmo é recolhido por uma *thread* que o atribui à máquina com menos memória disponível, memória essa que é obviamente superior ou igual à do *job*.

Viciação da Prioridade

O parâmetro que mais pesa na prioridade de um *job* é a sua tolerância, contudo esse valor é inserido pelo próprio utilizador, assim sendo, numa utilização incorreta do programa, se todos os utilizadores definirem que os seus *jobs* têm tolerância zero, o algoritmo deixará de fazer sentido e a fila de espera passará a ser um *fifo*.

Uma forma de evitar este problema seria atribuir uma tolerância constante a todos os *jobs*, todavia julgámos que seria mais interessante admitir tolerâncias variáveis que fossem definidas pelos próprios utilizadores, desta forma cada um deverá ser minimamente altruísta para não sabotar o algoritmo de escalonamento.

Pacotes

Na comunicação entre o cliente e o servidor, passam dados cuja natureza varia constantemente, uns são referentes a *login*, enquanto que outros podem ser simplesmente um pedido de consulta do estado do servidor.

Tendo isto em consideração, se encapsularmos todas as informações que pretendemos enviar num único pacote, as leituras e escritas no *socket* ficam significativamente mais simples, visto que abstratamente é serializada apenas uma estrutura de dados.

Ainda dentro deste tema, é necessário assegurar que o significado dos dados transmitidos não ser perde pelo simples facto de terem sido encapsulados, como tal foi criada uma pequena hierarquia de classe que ajuda a resolver esse problema.

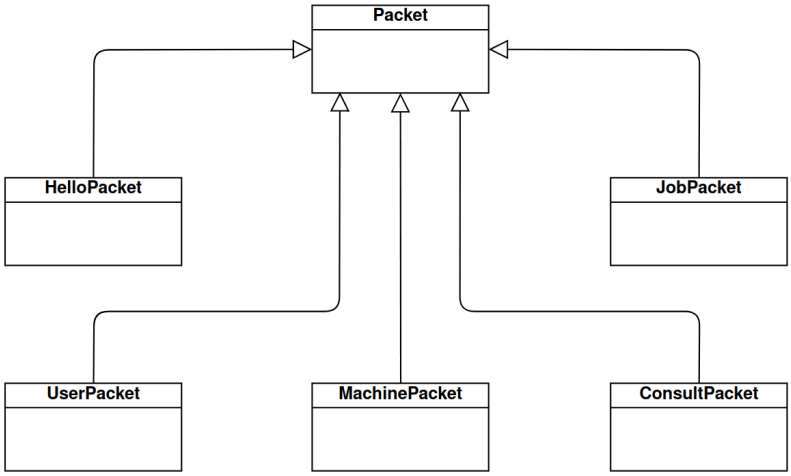

Figura 6. Hierarquia de classes de um *packet*

A partir desta construção, definimos que os dados referentes a um utilizador são encapsulados num *UserPacket*, enquanto que os das máquina são inserido num *MachinePacket*, consequentemente é possível conhecer a finalidade dos dados sem que estes tenham sido sequer analisados.

Dentro de cada pacote existem ainda protocolos específicos que ajudam a identificar outros tipos de informações, contudo o essencial desta hierarquia reside no facto de apenas circular uma classe abstrata através dos *sockets*, o que facilita imenso a serialização e deserialização de objetos.

Decisões Tomadas

Durante a realização do trabalho prático, verificámos que o enunciado mencionava alguns pontos onde as explicações eram poucas ou quase nenhumas, o que nos obrigou a tomar algumas decisões que em muito influenciaram o estado final do projeto.

Diferenciação entre Utilizadores e Máquinas

Quando uma determinada entidade ser conecta ao servidor, não sabemos se estamos perante um utilizador ou uma máquina, como tal definimos que a primeira mensagem a ser recebida pelo servidor é um identificador que permite o redirecionamento do seu *worker* para as operações relativas à entidade em questão.

Outra forma de resolver este problema seria atribuir uma porta distinta a cada entidade, contudo a solução anterior parece-nos mais adequada.

Adição e Remoção de Máquinas

A nossa implementação permite a inserção de novas máquinas enquanto o servidor está a atribuir *jobs*, por outro lado, a partir do momento em que uma máquina se apresenta ao servidor, esta jamais o poderá abandonar.

A princípio procurámos implementar a possibilidade de uma máquina sair do sistema, mas depois percebemos que isso iria adicionar um novo grau de complexidade ao trabalho prático, algo que não desejávamos.

Granularidade dos Locks

Todos os *locks* e *conditions* que utilizámos estão implementados ao nível das estruturas de dados, o que se torna bastante inconveniente no sentido em que acabámos por acordar *threads* que deveriam manter-se no estado de *sleep*.

Por outro lado, se a granularidade estivesse ao nível dos clientes e das máquinas, haveria uma quantidade exorbitante de *locks* e *conditions*, o que atrasaria certamente a execução do servidor e acrescentaria mais complexidade ao código.

Tendo isto em consideração, e sabendo que os serviços de *Cloud Computing* são utilizados por milhares de utilizadores, optámos por implementar uma granularidade mais superficial.

Conclusão

Durante a realização deste trabalho prático enfrentámos vários desafios que nos obrigaram a pensar e refletir qual seria a melhor estratégia a adotar. Nunca vimos estas dificuldades como algo maçador, muito pelo contrário, procurámos sempre dar o nosso melhor e seguir em frente a fim de apresentar um projeto do qual nos pudéssemos orgulhar.

Desta forma, acreditamos que cumprimos com o nosso objetivo, para além de termos implementado devidamente as funcionalidades básicas, as funcionalidades avançadas também apresentam o comportamento esperado.

É certo que nalguns momentos pensámos em implementar novos serviços, como por exemplo a inserção e remoção de máquinas, contudo a falta de tempo levou-nos a fazer escolhas que impossibilitaram a realização desses serviços na íntegra.

Em suma, este trabalho prático serviu para consolidar alguns conceitos que ficaram pouco claros numa primeira abordagem, mas mais que isso, abrir-nos os olhos para a complexidade de escalonar pedidos sem que alguém fique bloqueado ou sofra de *starvation*.