Sistemas Distribuídos

Projecto:

Sistema de Mercado Distribuído para Job Shop Scheduling (JSS) Colaborativo

Rui S. Moreira & Ivo Pereira

UFP - FCT

Março 2021

Version 2.1

1. Sistema de Mercado Distribuído para *Job Shop Scheduling* Colaborativo

1.1. Introdução

Este projecto tem dois objectivos principais: i) enriquecer os conhecimentos e a familiaridade dos alunos em relação aos vários aspectos e requisitos vulgarmente associados a projectos de sistemas distribuídos; ii) melhorar o entendimento e a prática dos alunos em relação à especificação e desenvolvimento de sistemas distribuídos.

Neste projecto pretende-se desenvolver uma solução distribuída para problemas de *Job Shop Scheduling* (JSS). Este é um problema de optimização combinatória NP-Completo[1], onde um conjunto de tarefas são atribuídas a sucessivos recursos para serem executadas. Numa versão padrão do problema, existem *J* **tarefas (*jobs*)** $\{J_1, J_2, ..., J_J\}$. Dentro de cada tarefa, há um conjunto *O* de **operações** $\{O_1, O_2, ..., O_O\}$ que precisam ser processadas numa ordem específica (com restrições de precedência) em *M* **máquinas** $\{M_1, M_2, ..., M_m\}$. Cada operação tem uma **máquina** específica na qual precisa ser processada e apenas uma operação de cada tarefa pode ser processada num determinado momento. Por outras palavras, há um número *J* de tarefas, cada uma com um número *O* de operações que devem ser executadas sequencialmente num número *M* de máquinas (ver exemplo na Tabela 1).

Tabela 1 - Sequência de máquinas para executar as operações de cada tarefa

	O1	**O2**	**O3**
J1	M2	M3	M1
J2	M1	M3	M2
J3	M3	M2	M1

[1] https://pt.wikipedia.org/wiki/NP-completo

A combinação da execução de todas as *O* operações, de todas as *J* tarefas, em todas as *M* máquinas, resulta num plano de escalonamento (*scheduling*). Normalmente o plano de *scheduling* é representado através de um diagrama de Gantt, com um determinado tempo mínimo para concluir todo o processo. Neste projecto, o objectivo da resolução do problema de JSS será minimizar este tempo de conclusão, também conhecido como *makespan*[2].

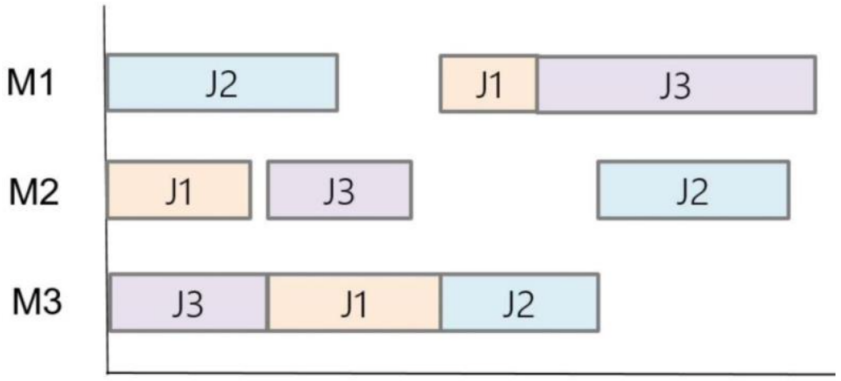

Figura 1 - Diagrama de Gantt com uma sequência possível de execução das tarefas nas máquinas

A Figura 1 representa um escalonamento possível (*schedule*) da execução das operações das tarefas da Tabela 1 nas diferentes máquinas. Neste exemplo, a tarefa J2 é executada na máquina M1 e, paralelamente, as tarefas J1 e J3 são executadas nas máquinas M2 e M3, respectivamente. Após a sequência inicial ser concluída, é possível reparar que a máquina M2 concluiu a execução da tarefa J1 e estaria disponível para iniciar a execução da próxima operação. No entanto, a próxima operação agendada pertence à tarefa J3, que ainda está a ser executada na máquina M3 e, portanto, a máquina M2 deve esperar até que a execução da tarefa J3 na máquina M2 seja concluída. Todas as restantes execuções de operações das tarefas são processadas de maneira semelhante, obedecendo às restrições de precedência/sequenciamento entre operações de cada tarefa.

[2] https://en.wikipedia.org/wiki/Makespan

1.2. Descrição e Objectivos do Projecto

Neste projecto, propõe-se o desenvolvimento de um sistema de mercado distribuído que permita a vários clientes/utilizadores registarem-se no serviço (Figura 2). Posteriormente os utilizadores podem iniciar sessões de trabalho (*session*), através das quais podem listar e criar grupos de tarefas partilhadas (***job groups***). Cada *job group* serve para que o seu criador/dono submeta uma tarefa de *scheduling*. Para o fazer deve carregar no sistema um conjunto de créditos suficiente para a execução da tarefa, bem como os dados referentes ao JSS específico para o qual se pretende encontrar a melhor solução possível. Os outros utilizadores podem associar-se a uma tarefa de um *job group*, participando na sua execução, através de um ou mais *workers/schedulers*, obtendo com isso créditos. Os *workers* ganham créditos associados à identificação/obtenção de planos de *scheduling* possíveis para o *job group* a que se associaram.

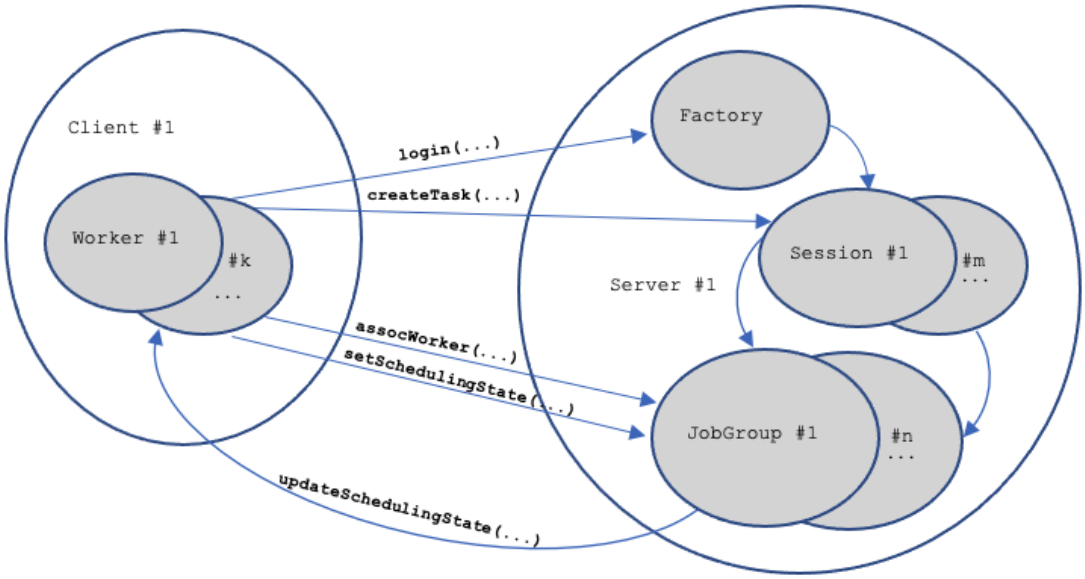

Figura 2 - Esquema simplificado de instâncias do sistema distribuído.

Cada *job group* permitirá então dividir por vários trabalhadores (*workers*), de forma colaborativa, a tarefa de obtenção de vários planos de *scheduling* possíveis, para o problema específico de JSS criado pelo dono do *job group*.

Em cada *job group* pretende-se aplicar algoritmos de optimização para a obtenção de planos de *scheduling*. Para isso, cada *worker* deverá aplicar exaustivamente um algoritmo de optimização pré-seleccionado na criação do *task group*, tal como *Tabu Search*[3] (TS) ou *Genetic Algorithm*[4] (GA), com recurso ao código auxiliar que é fornecido e detalhado na Secção 1.5.

Os *workers* deverão receber 1 crédito por cada instância possível de JSS encontrada e 10 créditos se encontrarem a melhor solução de entre todas. Os créditos devem ser retirados ao *plafond* do dono do *job group*, à medida que forem sendo atribuídos aos utilizadores pelo trabalho dos respectivos *workers*.

Aquando da criação do *job group*, o utilizador deverá então carregar os créditos necessários para pagar os planos de *scheduling* que forem encontrados pelos vários workers. Adicionalmente, o dono do *job group* deve especificar toda a informação necessária para a definição do problema JSS, e.g. através de um ficheiro contendo:

i) a instância de problema JSS a resolver (cf. matriz com o conjunto de tarefas, máquinas e sequência de operações com tempos respectivos em cada máquina). São disponibilizadas 53 instâncias de JSS da OR-Library[5] na pasta `edu/ufp/inf/sd/project/data`;

ii) o algoritmo de optimização a utilizar (TS ou GA);

iii) a definição da estratégia evolutiva do GA, a ser alterada por parte do coordenador *job group* ao longo do tempo.

1.3. Descrição dos Requisitos Concretos do Projecto

Dada a descrição genérica do problema de JSS, pretende-se de seguida detalhar mais pormenorizadamente os requisitos concretos deste projecto:

R1. O sistema deverá aceitar o registo de vários utilizadores/clientes, que devem indicar o *username* e a *password*. Posteriormente, os clientes podem efectuar *login* para

[3] https://en.wikipedia.org/wiki/Tabu_search
[4] https://en.wikipedia.org/wiki/Genetic_algorithm
[5] http://people.brunel.ac.uk/~mastjjb/jeb/orlib/jobshopinfo.html

obterem uma sessão de trabalho, na qual poderão efectuar operações básicas de gestão de *job groups* (e.g. *list job groups*, *create job group, pause job group, delete job group*, etc.).

R2. Cada *job group* corresponderá a uma "reunião/sala" à qual vários *workers* (de diferentes utilizadores) podem ser associados, para colaborar na tarefa de encontrar planos de *scheduling*. Cada utilizador poderá associar um ou mais *workers* ao mesmo *job group*. A associação de um *worker* permite que lhe seja atribuída uma instância de JSS para tratar. As instâncias distribuídas pelos *workers* deverão ser exclusivas, para evitar *overlapping* e duplicação de esforços.

R3. Numa primeira fase, a distribuição de tarefas pelos vários *workers* deverá seguir o padrão `Observer`, implementado com recurso ao RMI. Nesta abordagem, os *workers* (*observers*) podem-se registar num *job group* (*subject*) para o qual vão trabalhar. Nesta fase, os *workers* deverão utilizar o algoritmo de pesquisa *Tabu Search* (TS). O *job group* deverá atribuir aos vários *workers* a informação/parametrização base necessária, para estes executarem as suas instâncias do algoritmo de pesquisa TS. Neste caso, o *job group* deverá receber todos os possíveis *schedules* encontrados pelos vários *workers*, contabilizar os créditos e no final informar os *workers* da melhor decisão.

R4. Numa segunda fase, a distribuição de tarefas pelos vários *workers* deverá seguir o padrão `Publish/Subscribe`, com recurso ao RabbitMQ. Nesta abordagem, os *workers* deverão utilizar um *Genetic Algorithm* (GA) para permitir pesquisar soluções de *scheduling* além dos mínimos locais (como é feito pelo TS). O GA deverá permitir ao coordenador *job group* enviar alterações à evolução das pesquisas ao longo do tempo. Cada *worker* deverá ter um ID único sequencial, gerado pelo coordenador. Este ID poderá ser usado para nomear as filas de comunicação locais entre os *workers* (em Java) a instância do GA (em Python).

As estratégias dos requisitos R3 e R4 deverão ser implementadas de forma evolutiva, ou seja, começando por implementar a primeira (em RMI) e só depois passar à segunda (com RabbitMQ) que é mais complexa. Em qualquer uma das estratégias, os *workers* deverão receber 1 crédito por cada instância de *Job Shop Scheduling*

executada e 10 créditos se encontrarem a melhor solução de entre todas. Os créditos devem ser retirados ao *plafond* que o dono colocou no *job group*, à medida que forem sendo atribuídos aos *workers* dos respectivos utilizadores

O sistema deverá ainda permitir gerir a propagação das atualizações em ambas as estratégias e de acordo com os padrões utilizados (cf. `Observer vs. Publish/Subscribe`). Sempre que for encontrada uma solução, todos os *workers* deverão ser informados. Assim que terminar a pesquisa (e.g. por tempo máximo, por número máximo de soluções, por plafond máximo disponível, etc.), todos os *workers* devem ser informados para pararem a sua execução.

Serão valorizados os projectos que tirem partido da utilização de ***threads*** para melhorar a arquitectura e eficiência do código (tanto no servidor como nos clientes), bem como tirem partido das arquitecturas *multicore* existentes actualmente. Deverão também identificar e implementar, sempre que necessário, a respectiva sincronização destas *threads* no acesso a regiões críticas partilhadas.

R5. O sistema deve ainda garantir a tolerância a falhas, ou seja, se o servidor ou um dos nós falhar, então os restantes nós devem assegurar a continuidade do serviço, dos *job groups* e respectivos JSS a analisar.

R6. Deverão ainda ser consideradas questões de segurança, tanto na autenticação como na partilha de conteúdos, de modo a controlar acessos indevidos aos recursos partilhados. Numa primeira fase pode considerar-se autenticação em *plain text* e numa segunda fase considerar a utilização de autenticação baseada em *JSON Web Token* (JWT)[6].

R7. Será necessário desenvolver uma interface gráfica para que os clientes possam usar o sistema e gerir todo o processo de criação e associação aos grupos de JSS, bem como visualizar os resultados dos vários processos. Devem ainda fazer uma avaliação comparativa, para vários problemas JSS a fornecer, e planear uma demonstração do sistema desenvolvido para 15 minutos.

[6] URL: https://jwt.io/

De acordo com o descrito, o principal objectivo será o desenvolvimento do serviço de partilha de processamento, de forma distribuída. O sistema deve facilitar a organização e a coordenação remota dos vários grupos de *workers* que acedem de forma concorrencial aos mesmos recursos. Paralelamente devem considerar os aspectos de tolerância a falhas e as questões de segurança. Os requisitos serão pesados de acordo com a Tabela 2.

Tabela 2 - Pesos atribuídos a cada requisito

Requisitos	R1	R2	R3	R4	R5	R6	R7
Cotação	3	2	4	5	2	2	2

1.4. Ferramentas Tecnológicas a Utilizar

Há vários padrões de design (***design patterns***[7]) que podem ajudar na organização da estrutura e do comportamento da aplicação distribuída solicitada. Por exemplo, o *observer design pattern* é muito utilizado para gerir sistemas com comunicação *publish/subscribe*. Este padrão permite aos clientes subscrever certos assuntos/eventos para posterior notificação. Outro exemplo, o *factory method design pattern*, é também uma solução adequada para criar sessões ou outros tipos de objectos, de acordo com o perfil dos utilizadores.

O sistema proposto deve ser implementado recorrendo a tecnologias e ferramentas de middleware oferecendo comunicação síncrona (e.g. RMI) e comunicação assíncrona (e.g. RabbitMQ). **É obrigatório utilizar tanto comunicação síncrona como comunicação assíncrona.**

Os alunos são convidados a explorar soluções arquitecturais e de *design* que melhor se coadunem à implementação dos algoritmos de gestão, coordenação e sincronização da aplicação distribuída proposta. Devem dar atenção especial às questões de autenticação, sincronização, coordenação, tolerância a falhas e segurança. Sugere-se que utilizem soluções de design e algoritmos semelhantes aos abordados nas aulas.

[7] URL: http://pages.cpsc.ucalgary.ca/~kremer/patterns/; URL: http://www.fluffycat.com/java/patterns.html

1.5. Explicação do código auxiliar

MUITO IMPORTANTE:

Deverão instalar o RabbitMQ[8] e instalar o package *pika* do Python: `python3 -m pip install pika --upgrade`

No IntellJ deverão adicionar as bibliotecas do RabbitMQ Java Client 5.11.0[9] e as dependências SLF4J API 1.7.30[10] e SLF4J Simple 1.7.30[11]. Podem ir a ***"File -> Project Struture"*** e nas **Libraries** adicionar a partir do Maven (Figura 3):

```
com.rabbitmq:amqp-client:5.11.0
org.slf4j:slf4j-simple:1.7.30
```

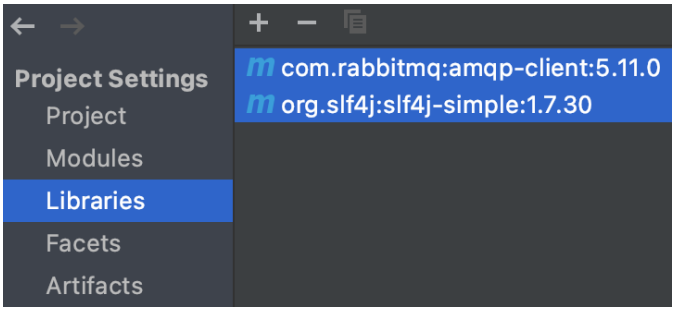

Figura 3 - Estrutura de ficheiros do *package* `edu.ufp.inf.sd.project`

Alternativamente podem fazer download dos JARs, colocar numa pasta lib dentro do projeto e fazer a associação direta.

É fornecido um *package* `edu.ufp.inf.sd.project` configurado com uma estrutura similar aos exercícios das aulas práticas (Figura 4). Na pasta *client* deverá ficar todo o código do lado do cliente. Na pasta *consumer* está um exemplo de um consumidor de filas *RabbitMQ*. Na pasta *data* estão 53 instâncias do JSS da OR-Library, que poderão usar para este trabalho. Na pasta *producer* está um exemplo de um produtor de filas *RabbitMQ* para mudança de estratégias de cruzamento do GA. Nas pastas *runscripts_rmi*

[8] https://www.rabbitmq.com/download.html
[9] https://mvnrepository.com/artifact/com.rabbitmq/amqp-client/5.11.0
[10] https://mvnrepository.com/artifact/org.slf4j/slf4j-api/1.7.30
[11] https://mvnrepository.com/artifact/org.slf4j/slf4j-simple/1.7.30

e *runscripts_rmq* estão os scripts relativos ao *Java RMI* e ao *RabbitMQ*, respectivamente (devem atualizar os ficheiros *setenv* dentro de cada pasta para usar no vosso computador). Na pasta *server* deverá ficar todo o código do lado do servidor. Na pasta *util* são fornecidos dois packages (cf. `edu.ufp.inf.sd.project.util.geneticalgorithm` e `edu.ufp.inf.sd.project.util.tabusearch`).

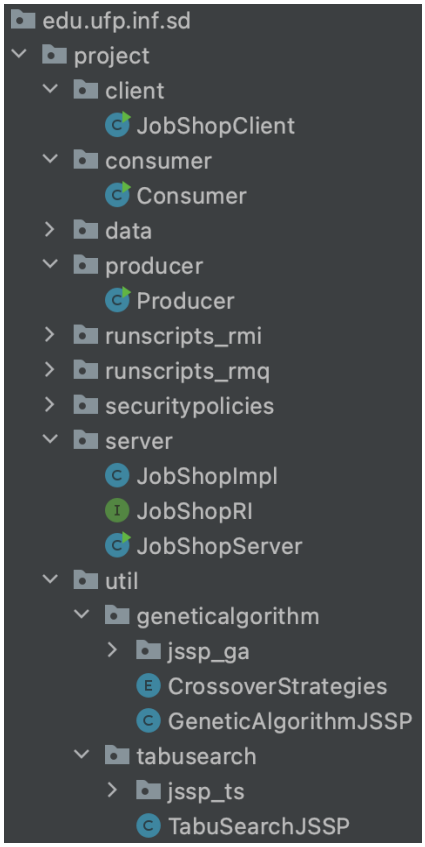

Figura 4 - Estrutura de ficheiros do *package* `edu.ufp.inf.sd.project`

1.5.1. Tabu Search

No package `edu.ufp.inf.sd.project.util.tabusearch` é fornecido o algoritmo TS em Python e uma classe `TabuSearchJSSP` que deverão usar para executar o algoritmo. Na pasta *server*, é apresentado um exemplo de utilização na implementação do serviço `int runTS(String)` na classe `JobShopImpl`. O projeto encontra-se configurado para executar uma instância remotamente, do cliente para o

servidor, apenas como demonstração. Deverão adaptar o código para cumprir os requisitos especificados. Na classe `JobShopClient` é executado o serviço `int runTS(String)`, como ilustrado na Figura 5.

```
//============= Call TS remote service ============
String jsspInstancePath = "edu/ufp/inf/sd/project/data/la01.txt";
int makespan = this.jobShopRI.runTS(jsspInstancePath);
Logger.getLogger(this.getClass().getName()).log(Level.INFO,
        msg: "[TS] Makespan for {0} = {1}",
       new Object[]{jsspInstancePath,String.valueOf(makespan)});
```

Figura 5 - Exemplo de execução do serviço `runTS`

> **Nota:** Na classe `TabuSearchJSSP`, poderão ter de alterar o comando Python acordo com o vosso Sistema Operativo, na linha 31. Sugere-se que coloquem o mesmo comando usado no ficheiro *_1_runpython*

1.5.2. Genetic Algorithm

No package `edu.ufp.inf.sd.project.util.geneticalgorithm` é fornecido o algoritmo GA em Python e uma classe `GeneticAlgorithmJSSP` que deverão usar para executar o algoritmo. É ainda apresentada uma enumeração `CrossoverStrategies` com três estratégias de cruzamento que o coordenador *job group* deverá utilizar para enviar alterações à evolução das pesquisas ao longo do tempo. As três estratégias de cruzamento são: (i) Dois indivíduos, escolhidos aleatoriamente, geram dois filhos que os substituirão na população; (ii) Dois indivíduos, escolhidos aleatoriamente, geram dois filhos que substituirão outros dois indivíduos aleatórios na população, sendo garantido que o indivíduo mais apto sobrevive (elitismo); (iii) Dois indivíduos, escolhidos aleatoriamente, geram apenas um filho que substituirá o pior indivíduo da população.

Nota: Na classe `GeneticAlgorithmJSSP`, poderão ter de alterar o comando Python acordo com o vosso Sistema Operativo, na linha 35. Sugere-se que coloquem o mesmo comando usado no ficheiro *_1_runpython*

O código Python do GA está preparado para começar a pesquisa com base numa estratégia definida inicialmente e comunicar os resultados para uma fila RabbitMQ. Além da instância e da estratégia de cruzamento a seguir (por omissão, será a `CrossoverStrategies.ONE`), é necessário enviar também uma fila para que o algoritmo fique à escuta de mudanças de estratégia de cruzamento e do comando "*stop*" para parar a execução. **É importante salientar que o algoritmo GA só parará de executar quando receber esta mensagem "*stop*".**

Na classe `JobShopClient` é apresentado um exemplo de configuração e execução do algoritmo GA, como ilustrado na Figura 6. Neste exemplo, é definida a fila "*jssp_ga*", que será a fila produtora de comandos para o algoritmo, e ainda a fila "*jssp_ga_results*" que será para onde o algoritmo enviará resultados. De notar que, do lado do Python, o sufixo "*_results*" é adicionado **automaticamente** à fila enviada. Neste exemplo, é definida a primeira estratégia de cruzamento, sendo expectável que esta seja alterada ao longo do tempo.

```
//============ Call GA ============
String queue = "jssp_ga";
String resultsQueue = queue + "_results";
CrossoverStrategies strategy = CrossoverStrategies.ONE;
Logger.getLogger(this.getClass().getName()).log(Level.INFO,
        msg: "GA is running for {0}, check queue {1}",
       new Object[]{jsspInstancePath,resultsQueue});
GeneticAlgorithmJSSP ga = new GeneticAlgorithmJSSP(jsspInstancePath, queue, strategy);
ga.run();
```

Figura 6 - Exemplo de execução do algoritmo GA

Na classe `Producer` encontra-se um exemplo de comunicação de mudanças de estratégias para o GA, bem como o envio da mensagem "*stop*" no final (Figura 7).

```java
// Change strategy to CrossoverStrategies.TWO
sendMessage(channel, String.valueOf(CrossoverStrategies.TWO.strategy));
Thread.currentThread().sleep( millis: 2000);

// Change strategy to CrossoverStrategies.THREE
sendMessage(channel, String.valueOf(CrossoverStrategies.THREE.strategy));
Thread.currentThread().sleep( millis: 2000);

// Stop the GA
sendMessage(channel,  message: "stop");
```

Figura 7 - Exemplos de envio de comunicações para o algoritmo

Neste exemplo, a classe `Consumer` é responsável por consumir as mensagens enviadas para a fila "*jssp_ga_results*". Na Figura 8 é apresentado um exemplo de conteúdo desta comunicação, sendo visível os resultados obtidos pelo algoritmo GA e a mudança de estratégias, bem como a notificação que o algoritmo irá parar a execução como resultado de ter recebido a mensagem *"stop"*.

```
[x] Received 'Setting Strategy 1'
[x] Received 'Makespan = 760'
[x] Received 'Makespan = 758'
[x] Received 'Makespan = 755'
[x] Received 'Makespan = 726'
[x] Received 'Makespan = 724'
[x] Received 'Makespan = 720'
[x] Received 'Makespan = 709'
[x] Received 'Makespan = 706'
[x] Received 'Setting Strategy 2'
[x] Received 'Makespan = 704'
[x] Received 'Setting Strategy 3'
[x] Received 'Makespan = 680'
[x] Received 'Makespan = 679'
[x] Received 'Makespan = 671'
[x] Received 'Makespan = 669'
[x] Received 'Makespan = 666'
[x] Received 'Stopping...'
```

Figura 8 - Exemplo de comunicação para a fila "*jssp_ga_results*"

Os alunos deverão analisar o código e adaptar para conseguir cumprir os requisitos especificados.

2. Grupos, Agenda e Relatórios

Os alunos devem organizar-se em grupos de 3 elementos. Devem planear e dividir bem o trabalho de modo a que todos participem na elaboração do mesmo.

Estão previstos dois momentos de entrega que serão agendados na plataforma de *elearning*. Numa primeira fase deverá ser entregue um ficheiro com os diagramas UML de classes bem como os diagramas de sequências de mensagens. Numa segunda fase deverão ser entregues os diagramas UML finais e a implementação efectuada (código) com a respectiva documentação (gerada a partir dos comentários no código).

Deverão adoptar uma arquitectura modular, faseando o trabalho e permitindo uma implementação incremental, solucionando inicialmente os problemas mais fáceis e posteriormente resolvendo as situações mais complexas. Devem focar-se nos aspectos essenciais: arquitectura de distribuição, autenticação, sincronização e propagação de actualizações, tolerância a falhas e segurança.

No final devem entregar num único ficheiro ZIP todo o software desenvolvido (src), a documentação gerada e os respectivos diagramas UML.

Os alunos devem colocar e partilhar o seu projecto IntelliJ no Github[12] de modo a facilitar a colaboração no desenvolvimento, de todos os elementos do grupo. Devem ainda partilhar o projecto do Github com os professores através dos seus emails (ivopereira@ufp.edu.pt, rmoreira@ufp.edu.pt).

[12] Github: https://github.com/