Leonardo Santos Paulucio

Relatório do 2º Trabalho de Processamento Paralelo e Distribuído

Vitória - ES

05 de Julho de 2018

Leonardo Santos Paulucio

Relatório do 2º Trabalho de Processamento Paralelo e Distribuído

Trabalho apresentado à disciplina de Processamento Paralelo e Distribuído do curso Engenharia da Computação da Universidade Federal do Espírito Santo como requisito parcial de avaliação.

Professor: João Paulo A. Almeida

Universidade Federal do Espírito Santo

Engenharia da Computação

Vitória - ES

05 de Julho de 2018

Sumário

1 Introdução

Com o crescente desenvolvimento dos sistemas distribuídos nos últimos tempos surgiu a necessidade de serviços que independem de um acoplamento direto entre remetente e destinatário. Para que essa comunicação aconteça, se faz necessário o uso de um intermediário. Através dele é possível se obter duas características importantes: desacoplamento espacial e desacoplamento temporal.

O desacoplamento espacial permite que as entidades remetentes e destinatárias não precisam se conhecer. Isso permite que os desenvolvedores possam usar a tecnologia de desenvolvimento que acharem melhor. Como hoje existe uma gama enorme de ferramentas, essa propriedade fornece ao desenvolvedor um grau de liberdade imenso.

Já o desacoplamento temporal permite que as entidades possam ter tempos de vida independentes umas das outras, isso garante que elas não precisam estar ativas ao mesmo tempo para poderem realizar a troca de mensagens.

Todas essas características tornam a comunicação indireta uma ferramenta muito usada em ambientes móveis/WEB e ambientes voltados a ocorrência de eventos. Como exemplo, pode se citar os *Web Services*.

Os *Web Services* permitem uma integração de sistemas e na troca de informações entre diferentes aplicações, permitindo que sistemas criados em plataformas totalmente diferentes sejam compatíveis. Para isso, a aplicação usa uma linguagem universal para se comunicar através de um intermediário, atualmente pode se citar o XML, JSON, CSV e etc.

Esse trabalho tem por objetivo praticar programação paralela e distribuída usando a API Java JMS, além de realizar análises de desempenho em um cluster de computadores. Ele consistirá na implementação de uma arquitetura mestre/escravo para realizar um ataque de dicionário em uma mensagem criptografada utilizando troca de mensagens em duas filas.

2 Implementação

A implementação do trabalho foi feita utilizando a IDE NetBeans 8.2. Foram criados vários pacotes para facilitar a organização do código e das implementações de cada elemento. O trabalho é composto dos seguintes pacotes:

- **br.inf.ufes.ppd:** Nesse pacote estão as interfaces padrões com os serviços oferecidos pelo mestre, a classe *Guess* e a classe *SubAttackJob*.
- **br.inf.ufes.ppd.application:** Nesse pacote estão as aplicações propriamente ditas. Estão nesse pacote as aplicações de Cliente, Escravo e Mestre.
- **br.inf.ufes.ppd.implementation:** Nesse pacote se encontram as implementações das interfaces e das funcionalidades do mestre e escravo.
- **br.inf.ufes.ppd.utils:** Por fim, nesse pacote estão as funcionalidades de criptografia e descriptografia e a de geração de dados em ".csv"para geração dos gráficos.

Para o correto funcionamento da aplicação mestre, desenvolvida nesse trabalho, sempre é necessário adicionar a seguinte diretiva ao comando de inicialização do mesmo.

```
-Djava.rmi.server.hostname=(IP DA MAQUINA HOST)
```

Essa diretiva é necessária para que o Java RMI possa criar a referência correta ao exportar um objeto remoto.

Outro comando que deve ser executado antes da inicialização de qualquer elemento é o *rmiregistry*. Ele deve ser executado dentro da pasta raiz onde se encontram as classes, no caso do NetBeans essa pasta é "build/classes/". Nessa pasta se encontra a pasta raiz do pacote do trabalho, nesse caso a pasta "br".

2.1 Cliente

A aplicação cliente é a mesma implementada no 1º trabalho da disciplina e não foi alterada nenhuma parte do código. O programa cliente recebe como argumentos: o nome do arquivo criptografado, o trecho conhecido do texto original, um terceiro parâmetro que indica o tamanho do vetor de *bytes* aleatório que será gerado em caso do arquivo criptografado não existir, e por último o dicionário de chaves.

Assim, o cliente é responsável por localizar o mestre utilizando o *Registry*(utilizando a funcionalidade de *lookup*) e solicitar o serviço de ataque através do método *attack*. Ao

solicitar o serviço, o cliente passa o arquivo criptografado e o trecho conhecido, ficando bloqueado enquanto aguarda uma resposta do mestre.

Caso o nome do arquivo, fornecido como argumento para o programa cliente, seja inválido ele fica responsável por gerar um vetor aleatório de bytes, cujo tamanho será igual ao 3º parâmetro fornecido ou, caso esse não exista, será usado um tamanho aleatório na faixa de 1Kb a 100Kb.

O comando utilizado para iniciar o cliente é:

```
java br.inf.ufes.ppd.application.Client <Arquivo> <Trecho> <Tam.Vetor> <
    Dicionario>
```

2.2 Escravo

O escravo recebe como parâmetros: o caminho para o arquivo de dicionário, o nome do escravo que será criado e endereço do *host* onde está hospedado o *Glassfish*. Caso alguns desses parâmetros não sejam fornecidos ele pega seus valores padrões existentes em um arquivo de configurações.

Inicialmente, o escravo se conecta ao *Glassfish* que está rodando no *host* especificado para obter uma *Connection Factory*, e assim, as filas onde serão colocadas os subataques e os *guesses*. Após terminar essas tarefas de inicialização, o escravo fica ocioso esperando algum subataque ser adicionado na fila, para que, assim, possa iniciar o processamento.

Cada escravo realiza apenas um trabalho por vez, ou seja, quando existe um trabalho na fila o escravo o retira para realizar o processamento, e, somente quando esse trabalho é finalizado ele verifica na fila novamente se existe um novo trabalho para poder executar. Durante o processamento caso algum *Guess* seja encontrado o escravo o adiciona à fila de *Guesses* para que o mestre possa processar e montar a resposta que será enviada ao cliente.

O comando para se iniciar um escravo é:

```
java br.inf.ufes.ppd.application.SlaveServer <Dicionario> <Escravo> <Host>
```

É importante falar que para que a aplicação *SlaveServer* possa ser executada é necessário adicionar o arquivo *<gf-client.jar>* ao *classpath*. Caso esse arquivo já não esteja no diretório local deve se usar a seguinte diretiva na linha de comando(antes do nome da aplicação mostrado acima), para adicioná-lo:

```
java -cp .:../../glassfish5/glassfish/lib/gf-client.jar br.inf ...
```

2.3 Mestre

O mestre recebe como parâmetro o endereço do *registry* e do *host* onde está o *Glassfish*, caso eles não sejam fornecidos são utilizados os valores padrões existentes no arquivo de configurações.

O mestre é a aplicação que fornece o serviço de *attack* para um cliente. Quando esse serviço é solicitado ele cria vários subataques - o número de subataques depende de uma variável M que determina o tamanho do intervalo dos índices de cada trabalho - adicionando-os na respectiva fila, para que, os escravos possam pegar a tarefa e executar. O mestre fica sempre monitorando a fila de *Guesses*, através de uma *thread*, para processar os resultados obtidos pelos escravos de forma que seja possível montar a resposta ao final do ataque para o cliente. Quando todos os subataques são finalizados pelos escravos o mestre envia a resposta para o cliente com todos os resultados encontrados durante os subataques.

O comando para se iniciar um escravo é:

```
java br.inf.ufes.ppd.application.MasterServer <registry> <host>
```

Da mesma forma que foi explicado anteriormente para o escravo, é necessário adicionar o arquivo *<gf-client.jar>* ao *classpath* utilizando o comando:

```
java -cp .:../../glassfish5/glassfish/lib/gf-client.jar br.inf ...
```

Nas próxima seções serão discutidos os principais pontos da estrutura de dados utilizada no trabalho, decisões de projeto e problemas que ocorreram durante a implementação.

2.4 Estrutura de Dados

Como o cliente não precisou ser alterado, sua estrutura é a mesma do 1º trabalho. Ela é bem simples e não possui nada muito complexo já que sua função é, basicamente, localizar um mestre para solicitar o serviço de ataque enviando o arquivo criptografado e o trecho conhecido, após isso, fica bloqueado aguardando uma resposta do mestre. Sendo assim, sua estrutura não será discutida em detalhes.

O escravo sofreu alterações com a implementação feita no trabalho anterior. Sua estrutura ficou mais simples, visto que a operação de *checkpoint* e *rebind* não são mais necessárias. A única informação que o escravo possui é seu nome e uma lista de chaves candidatas, que são lidas de um arquivo de dicionário. Ao pegar um trabalho da fila ele percorre o intervalo de índices indicados na mensagem verificando se alguma dela produz uma mensagem com o trecho conhecido. Caso exista, ele produz um *Guess* e o adiciona a fila de *Guesses* para que o mestre possa pegar.

Com as alterações exigidas para o mestre, sua implementação também ficou mais simples, já que não se fez necessário realizar o gerenciamento de escravos e recuperação de erros. Basicamente o mestre recebe uma requisição de ataque de um cliente e cria vários *jobs* para esse ataque colocando-os na fila de subataques para que os escravos possam pegar. Quando uma requisição é recebida pelo mestre ele cria uma *thread* para gerar os *jobs*, isso permite que ele possa receber requisições em paralelo.

Para a implementação do mestre foi criada uma estrutura de controle de ataque chamada *AttackControl*. Essa estrutura é responsável por possuir as informações de um ataque solicitado por um cliente. Ele possui: a informação do tempo em que o ataque começou, possui uma variável que diz se o ataque está terminado, uma referência para a mensagem criptografada e para o trecho conhecido daquele ataque e por fim uma *HashMap* com os respectivos subataques e o *status* de cada um, indicando se o subataque já terminou ou não.

Para acessar as estruturas o mestre possui *HashMaps*, que facilitam o acesso e localização das mesmas. Essas *HashMaps* são as seguintes:

- **HashMap de AtaqueID em Lista de *Guess*:** Esse mapeamento permite que dado um número de ataque o mestre obtenha a lista de *guess* desse ataque.
- **HashMap de SubataqueID em AtaqueID:** Esse mapeamento permite ao mestre saber qual é o ataque que um subataque faz parte.
- **HashMap de AtaqueID em AttackControl:** Esse mapeamento permite que o mestre obtenha o *AttackControl* de um número de ataque.

Uma outra ferramenta utilizada na implementação do mestre é o *synchronized* de Java, que faz com que o acesso a uma determinada variável seja serializado. Essa ferramenta é de extrema importância para o mestre, visto que ele pode receber várias solicitações de ataque em paralelo, e o acesso a algumas estruturas se não forem serializados podem gerar exceção de acesso concorrente.

Para a troca de mensagens entre o mestre-escravo e escravo-mestre foi utilizado o *ObjectMessage*, que permite enviar um objeto em forma de mensagem, desde que esse objeto implemente o *Serializable* de Java, sendo assim, foram criadas duas classes para isso, são elas:

- **SubAttackJob:** Essa estrutura é utilizado na troca de mensagens do mestre para os escravos através da fila. É responsável por possuir as informações de um subataque criado pelo mestre, assim, possui: uma referência para a mensagem criptografada e para o trecho conhecido daquele ataque, o índice inicial e final que deve ser percorrido para aquele subataque, e por fim, o número do subataque.

- **Guess:** Essa estrutura é responsável pela troca de mensagens do escravo para o mestre. Ela possui a chave candidata, a mensagem decriptografada, o nome do escravo que encontrou a chave, o número do ataque e uma variável *booleana* que indica se aquele ataque acabou.

2.5 Decisões de Projeto

Durante a implementação do projeto algumas decisões tiveram que ser tomadas, elas estão discutidas a seguir.

1. **Lista separada de *guess***: A decisão de usar uma lista separada para os *guesses* foi basicamente por já ter sido usada e dado certo no 1º trabalho. Com essa estrutura fica simples o gerenciamento dos *guesses* obtidos, de forma que fica fácil montar a mensagem que será enviada ao cliente ao término do ataque.

2. **Criação de subataques:** Da mesma forma que para o item anterior, a criação de subataques foi usada no 1º trabalho, e se encaixou perfeitamente nesse segundo trabalho, permitindo uma grande facilidade no gerenciamento dos subataques criados para um dado ataque. Isso facilitou na forma de verificar se um ataque já tinha terminado, pois bastava verificar se todos os subataques de um ataque já tinham terminado.

3. **Fim de um Subataque:** Para determinar o final de um subataque foi usada a mesma classe *Guess* com os atributos vazios, com exceção do número do ataque e a variável *booleana* que representa o fim do ataque que era setada para *true*, dessa forma, quando o mestre pegava uma mensagem da fila a primeira coisa que ele faz ao desempacotar a mensagem é verificar a variável *booleana*, caso seja *false* é um *Guess*, caso contrário é uma mensagem indicando que o subataque acabou.

2.6 Problemas Durante a Implementação

Durante a implementação desse trabalho não foram encontrados muitos problemas. Foi possível aproveitar algumas estruturas que foram usadas no 1º trabalho e também os problemas que surgiram já eram conhecidos pois ocorreram durante a implementação do trabalho anterior.

Um novo problema que ocorreu relacionado ao JMS foi que quando não se tinha nenhum escravo rodando e o mestre recebia uma requisição de um cliente e gerava os respectivos subataques esses ataques ficavam na fila, assim, ao se iniciar uma aplicação de escravo ele conseguia pegar o trabalho da fila, porém ao se iniciar outro escravo o segundo não pegava mais nenhum trabalho. Isso ocorria pois existe uma funcionalidade

de *prefetch* do *Glassfish* que adiciona as mensagens a uma "cache"para cada escravo, porém ao se ajustar a configuração *imqConsumerFlowLimitPrefetch* esse problema foi resolvido.

Outro problema que surgiu foi para limpar as filas durante os testes de implementação, pois ao se fechar as aplicações as mensagens continuavam na fila, e ao reiniciar a aplicação do mestre ocorria um erro pois aquele ataque não existia. Para resolver isso foi feito um ajuste no código para limpar a fila antes de iniciar.

3 Análise de Desempenho

3.1 Máquinas e Equipamentos Utilizados

Para a análise de desempenho foram utilizados os computadores do laboratório de graduação (LabGrad I), que possuem as seguintes configurações: processador *AMD Athlon(tm) Dual Core Processor 5000B* com 4GB de memória RAM e o Sistema Operacional utilizado foi o *Ubuntu 16.04.4 LTS*. A configuração de rede das máquinas era o padrão Ethernet 100Mbps.

Durante as medições procurou-se deixar a máquina com apenas as aplicações rodando para evitar possíveis ruídos, mas é bom lembrar que existiam os processos em *background* do SO. Para a medição dos dados foi criada uma aplicação cliente para gerar tamanhos aleatórios de vetor de 0 a 50Kb, sendo que os tamanhos foram espaçados de 5 em 5Kb.

3.2 Cenário A

O cenário A consiste em 3 máquinas distintas, cada qual executando 4 aplicações de escravo, totalizando 12 escravos. A aplicação do mestre e do cliente foram executadas em uma outra máquina.

No dia dos testes para cenário A havia apenas um outro grupo realizando testes, e, dessa forma, não ocorreram muitos ruídos, diferentemente dos testes que serão mostrados para o cenário B.

A Figura 1 apresenta um gráfico com o tempo de resposta obtido com diferentes tamanhos de mensagem para o cenário A.

Analisando o gráfico, percebe-se, que à medida que o parâmetro M aumenta, o tempo de resposta também aumenta. Isso pode ser explicado pois o parâmetro M se relaciona com a faixa de índices que um trabalho irá possuir, e, por isso, também se relaciona com o número de trabalhos que são criados.

Um valor muito alto para M tende a produzir um número pequeno de trabalhos, porém com um intervalo grande para ser percorrido, o que acaba tornando o ataque muito próximo de uma execução sequencial. Por exemplo, com um M igual a 40000 e um intervalo de índices varia de 0 a 80000, seriam gerados dois trabalhos, um de 0 a 40000, e outro de 40000 a 80000. Assim, apenas dois escravos poderiam estar trabalhando no ataque, enquanto os outros ficariam ociosos pois não teriam trabalhos na fila para

executar. Dessa forma, o tempo de resposta tende a aumentar.

Já para valores menores de M, são criados vários trabalhos com intervalos de busca pequenos. Com isso, vários escravos podem pegar os trabalhos da fila e executar, diminuindo a ociosidade. Isso faz com que o ataque seja executado mais rapidamente, e, por consequência, o tempo de resposta tende a diminuir.

É possível notar também que os melhores tempos de resposta são obtidos com M de 5000 para baixo, e, ainda, que os tempos de resposta nessas faixas de M são muito próximas.

Apesar de no gráfico não ser possível de observar com clareza os tempos de resposta obtidos para os valores de M entre 50 e 400, por estarem sobrepostos, percebeu-se pelos testes que valores muito pequenos de M, abaixo de 300, possuíam um tempo de resposta maior. Isso pode ser explicado devido ao *overhead* existente na criação, processamento e troca de mensagens pela rede.

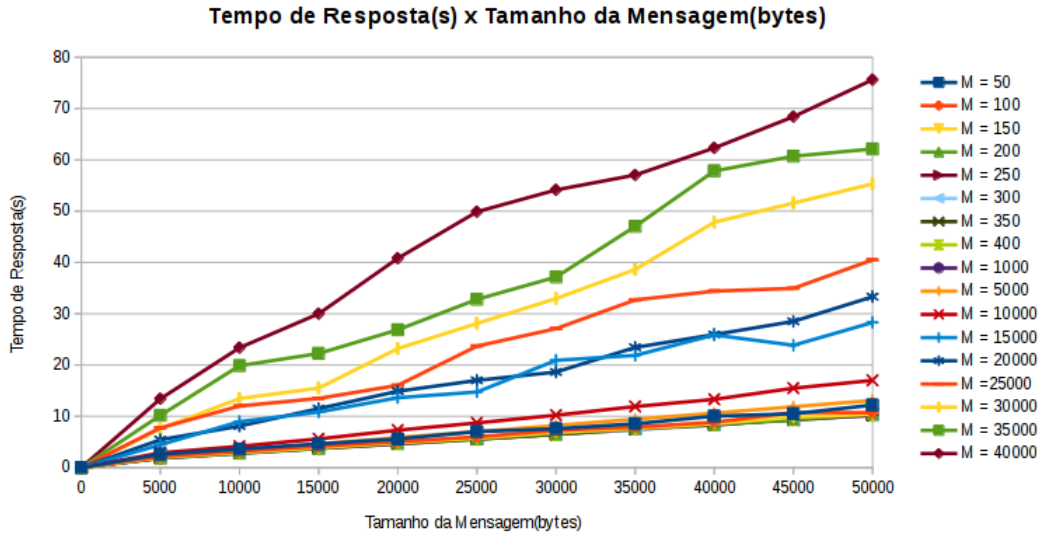

Figura 1 – Gráfico do Tempo de Resposta x Tamanho da Mensagem para o Cenário A

Através dos testes, percebeu-se que o melhor resultado obtido foi com M igual a 300. Foi feita uma comparação de desempenho entre o primeiro trabalho(rodando na mesma configuração de cenário) e o segundo trabalho com M igual a 300. O resultado está ilustrado na Figura 2.

No primeiro trabalho, o intervalo de índices que cada escravo deve executar é calculada através da divisão do tamanho total dos índices pelo número de escravos registrados, no caso dessa configuração existem 12 escravos.

A primeira coisa a se notar é que o desempenho do segundo trabalho, com M igual a 300, é muito melhor do que o primeiro, chegando a ser aproximadamente 4 vezes mais rápido. Isso pode ser explicado, para o trabalho 2, pelo fato de que para esse valor de M são gerados vários pequenos trabalhos, aumentando o nível de paralelismo, e, assim,

permitindo uma melhora no tempo de resposta. Já para o primeiro trabalho são produzidos exatamente 12 trabalhos, fazendo com que cada escravo tenha que processar 6670 índices aproximadamente.

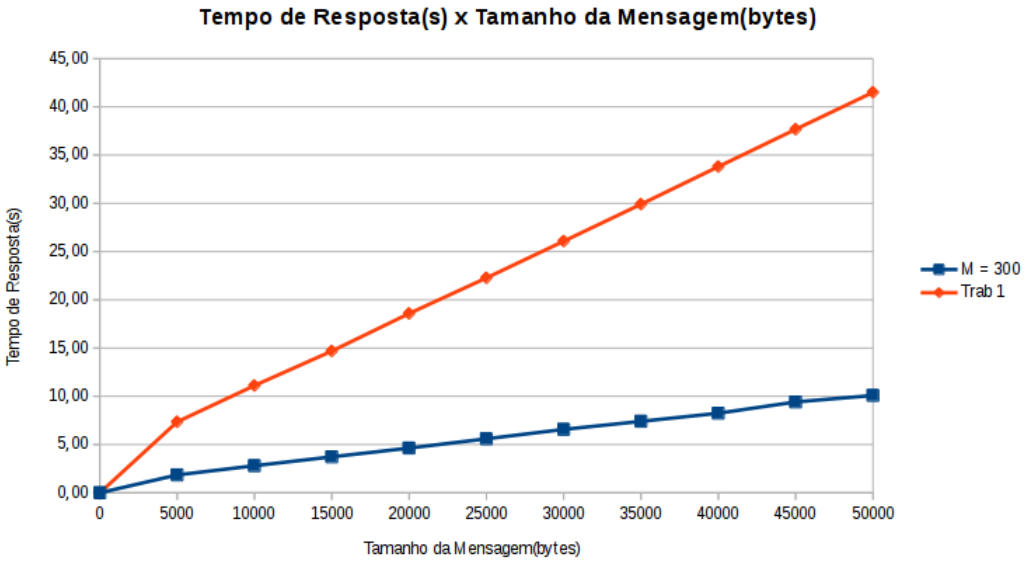

Figura 2 – Gráfico do Tempo de Resposta x Tamanho da Mensagem comparando o melhor M com o 1º trabalho - Cenário A

3.3 Cenário B

O cenário B, também consistia em 3 máquinas distintas, porém duas máquinas executavam 4 aplicações de escravo e uma executava 8, totalizando 16 escravos. A aplicação do mestre e do cliente também foram executadas em uma outra máquina. No dia dos testes do cenário B havia quatro grupos realizando testes, e, dessa forma, ocorreram algumas perturbações nos gráficos, porém elas não atrapalharam a realização das análises de desempenho.

A Figura 3 apresenta o gráfico dos resultados obtidos para o cenário B com vários valores de M.

Observando o gráfico, percebe-se que da mesma forma que ocorreu para o cenário A, à medida que o parâmetro M aumenta, o tempo de resposta também aumenta. Apesar de terem ocorridos algumas perturbações durante os testes, nota-se que o desempenho do segundo trabalho no cenário B é muito próximo do obtido no cenário A, mesmo com um computador sobrecarregado com 8 escravos.

Isso já era esperado, visto que, nessa segunda implementação o escravo era capaz de realizar o controle de consumo dos trabalhos, já que um novo trabalho só era obtido quando o atual tivesse terminado. Assim, no computador que está muito sobrecarregado, a execução do trabalho irá demorar, porém, da mesma forma, a obtenção de um novo

trabalho irá demorar, fazendo com que outros escravos que estejam disponíveis possam pegar o restante dos trabalho. Dessa forma, não ocorre perda significativa no desempenho.

Nota-se que para valores de M grandes, acima de 5000, o tempo de resposta aumentou quando comparado ao cenário A, e, também, pode se perceber oscilações para esses valores. Isso pode ter ocorrido pelo fato de que haviam 4 grupos realizando os testes no laboratório, e, como os testes eram executados na mesma rede, para mensagens maiores, o *overhead* de comunicação acabou aumentando.

Os melhores resultados para os tempos de resposta foram obtidos com M de 1000 para baixo, sendo que para esses valores os resultados foram muito próximos. Da mesma forma que ocorreu para o cenário A, não foi possível observar com clareza os tempos de resposta obtidos para esses pequenos valores, por ficarem sobrepostos no gráfico. Porém, pelos testes realizados os melhores resultados foram obtidos com M igual a 400.

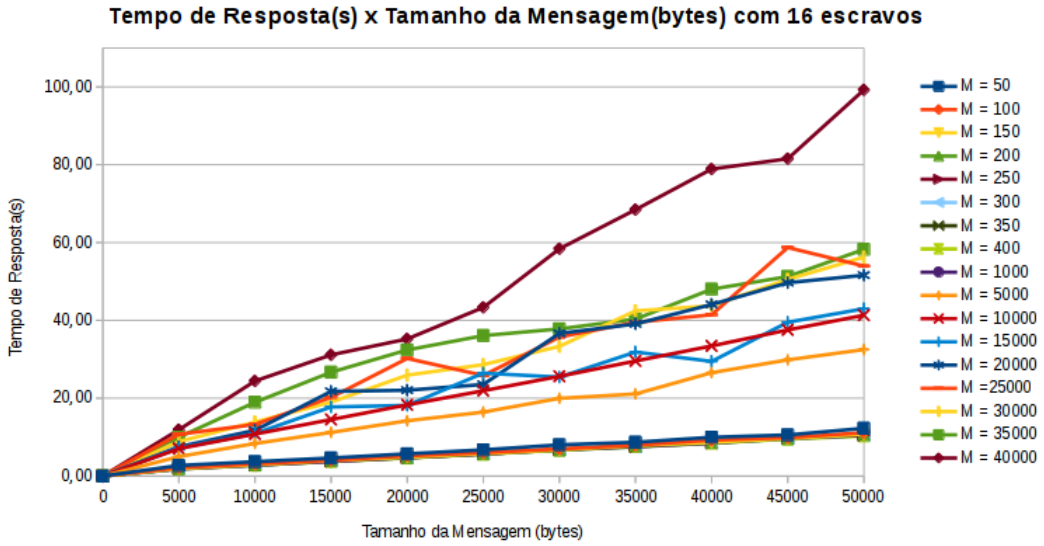

Figura 3 – Gráfico do Tempo de Resposta x Tamanho da Mensagem para o Cenário B

Também foi realizada uma comparação com a implementação do 1º trabalho na configuração do cenário B, o resultado está ilustrado na Figura 4.

É possível observar que o tempo de resposta da implementação do 2º trabalho é menor do que o 1º. Isso já era esperado visto que, existe uma máquina sobrecarregada executando 8 escravos. Na primeira implementação a divisão é feita igualmente entre todos os escravos, com isso, a máquina que possui 8 escravos recebe mais trabalhos do que as outras duas, nesse caso o dobro. Isso faz com que a máquina sobrecarregada demore ainda mais para executar os trabalhos, aumentando, assim, o tempo de resposta. Já no segundo trabalho esse problema não ocorre devido ao controle de consumo que o escravo é capaz de realizar.

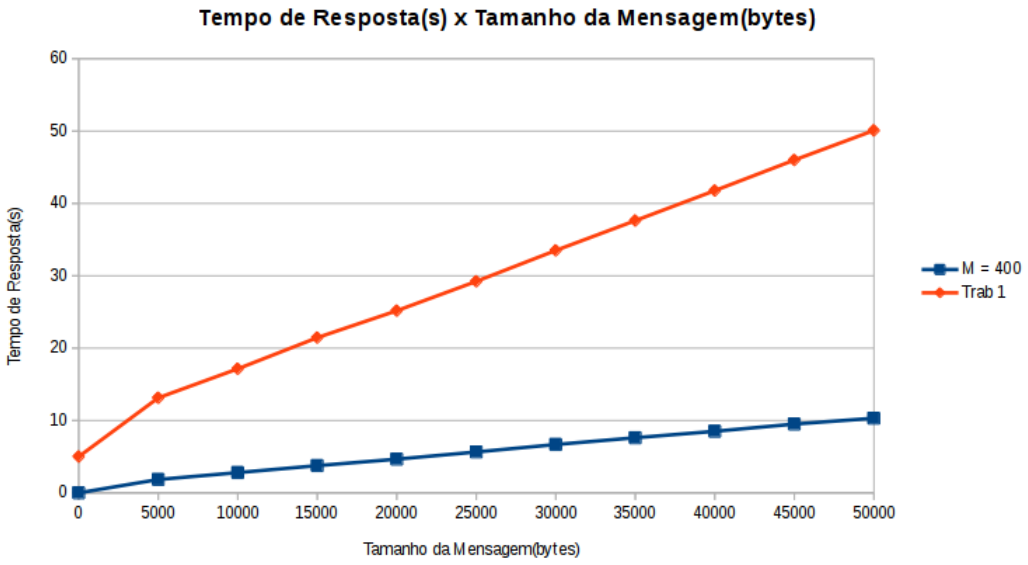

Figura 4 – Gráfico do Tempo de Resposta x Tamanho da Mensagem comparando o melhor M com o 1º trabalho - Cenário B

3.4 Comparação entre Cenários

A Figura 5 apresenta uma comparação dos tempos de resposta obtidos para o segundo trabalho utilizando os dois cenários de testes e escolhendo-se o melhor M. Para o cenário A foi utilizado M igual a 300 e para o cenário B foi usado M igual a 400.

Como já foi dito na seção anterior. É possível perceber que os cenários A e B possuíram resultados semelhantes. Mesmo com uma máquina sobrecarregada no cenário B, não houve um aumento do tempo de resposta. Isso já era esperado visto que nessa implementação ocorre o controle de consumo pelo escravo, dessa forma, a máquina mais sobrecarregada, executa menos trabalhos do que as outras, não compromentendo muito o tempo de resposta.

A Figura 6 apresenta uma comparação dos tempos de resposta obtidos para o primeiro trabalho utilizando os dois cenários de testes.

Nesse caso, já é possível perceber que no cenário B, onde existe uma sobrecarga em uma máquina, o tempo de resposta é maior do quando comparado ao cenário A. Como no primeiro trabalho não havia controle de consumo, a divisão era feita de forma igual entre os diversos escravos. Assim, a máquina sobrecarregada demorava para executar os trabalhos, fazendo com que o tempo de resposta aumentasse.

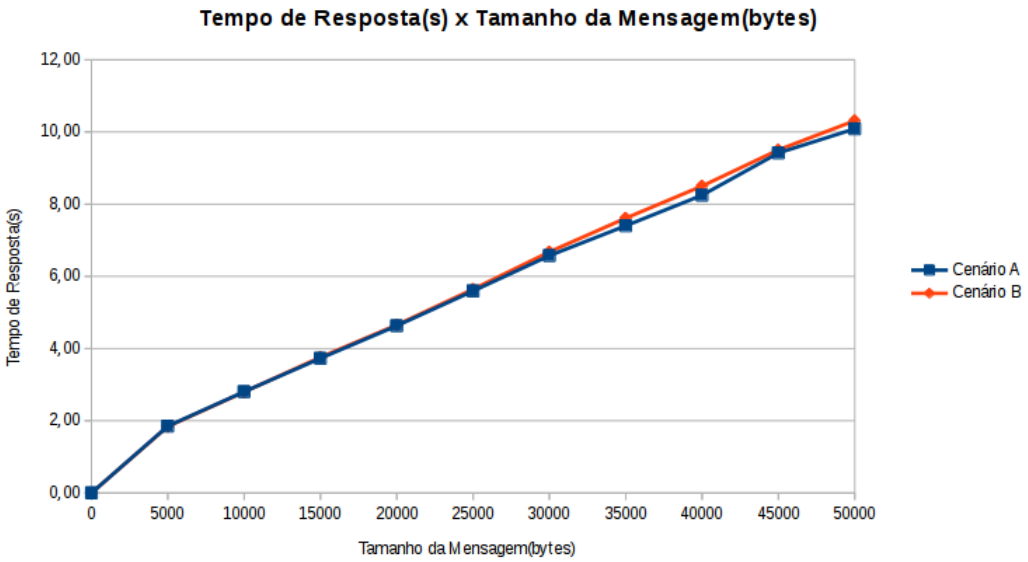

Figura 5 – Gráfico do Tempo de Resposta x Tamanho da Mensagem para o melhor M em Diferentes Cenários

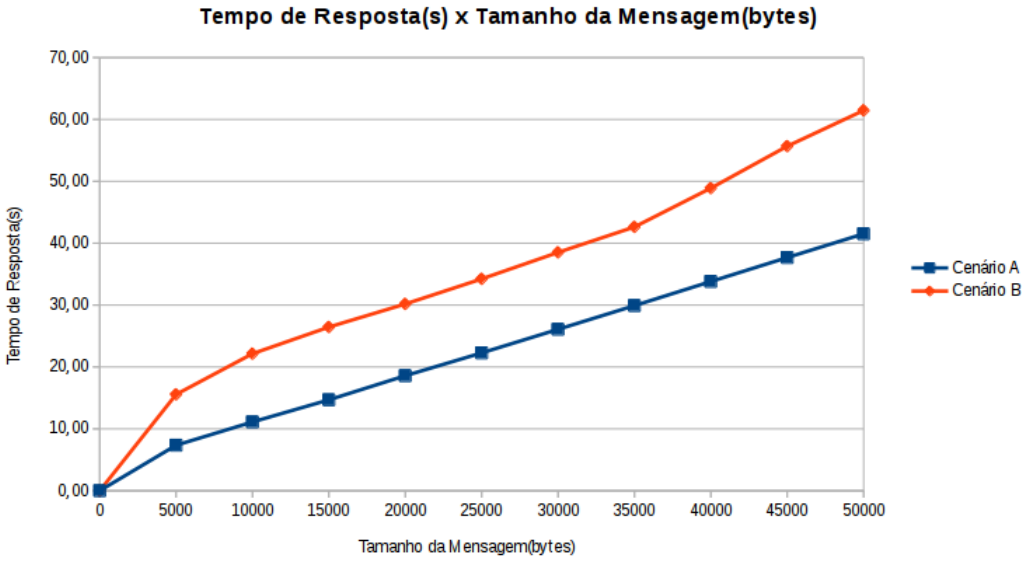

Figura 6 – Gráfico do Tempo de Resposta x Tamanho da Mensagem para o 1º trabalho em Diferentes Cenários

4 Conclusão

Ao final do trabalho, foi possível perceber as vantagens de se utilizar troca de mensagens em uma aplicação distribuída.

Os resultados obtidos mostraram que mesmo em um cenário onde uma máquina está sobrecarregada o tempo de resposta não é muito afetado devido ao controle de fluxo proporcionado por esse tipo de implementação. Diferentemente, no primeiro trabalho, isso não ocorre, e assim, em um cenário onde há uma sobrecarga em uma máquina, o tempo de resposta é afetado consideravelmente.

Algumas vantagens puderam ser notadas nesse segundo trabalho ao se utilizar troca de mensagens. A primeira é o desacoplamento espacial obtido, onde as entidades, nesse caso mestre e escravo, não precisavam conhecer umas as outras.

Outra vantagem é o desacoplamento temporal, que permite ao escravos poderem chegar e sair sem alterar o funcionamento do sistema. Isso fez com que a interface de *SlaveManager* pudesse ser completamente removida, visto que, o gerenciamento de escravos não era mais necessário.

Outro ponto interessante que pode se destacar é a simplicidade de implementação do mestre para esse segundo trabalho, quando comparado ao primeiro.

Apesar das vantagens citadas, existe um ponto negativo que pode ser notado ao se utilizar esse tipo de solução. Como a comunicação realizada de forma indireta, o mestre não possui informação nenhuma sobre a existência ou não de escravos. Dessa forma, caso ele receba uma requisição de um cliente, a única coisa que ele irá fazer é criar os subataques e esperar, até que todos estejam terminados para enviar uma resposta ao cliente. No primeiro trabalho, o mestre possuía uma lista com os escravos ativos, e caso, ele recebesse uma solicitação de ataque ele poderia verificar se essa lista estava vazia ou não, caso estivesse, poderia traçar alguma estratégia para finalizar o ataque, seja lançando uma exceção para o cliente ou realizando por si mesmo o ataque.