Implementação do algoritmo de consenso Raft

Diogo Santos, João Magalhães, Nuno Fernandes

Sumário : Neste trabalho foi implementado o algoritmo de consenso Raft, numa abordagem de programação orientada a objetos em Java. Baseamos esta implementação no artigo "In Search of an Understandable Consensus Algorithm"de Diego Ongaro e John Ousterhout da Universidade de Stanford [1]. Após efetuar uma introdução ao algoritmo, explicamos o funcionamento da nossa implementação de Leader Election, de Log Replication e finalmente caracterizamos a aplicação e efetuamos as medições que achamos necessárias.

***Keywords*—Raft, Líder, Seguidor, Log, Cluster, Cliente, Servidor, Term, Método, Classe, Objeto, Timeout**

I. Introdução

No contexto de sistemas de computação distribuída, onde vários processos sujeitos a falhas executam simultaneamente, muitas vezes em computadores distanciados fisicamente, e que trocam informação também sujeita a atrasos e perdas, surgiu a necessidade de implementar algoritmos, como algoritmos de consenso, que garantem a fiabilidade do sistema. Algoritmos de consenso ditam o comportamento dos vários processos envolvidos no sistema de maneira a que seja possível garantir concordância quanto ao valor de uma variável comum aos processos, na identidade do líder numa rede de computação distribuída, concordância em realizar uma tarefa que deve ser executada por todos os processos, entre outros.

Numa grande parte de problemas de consenso em sistemas distribuídos recorre-se a soluções implementando Paxos ou variações desse algoritmo [1]. O algoritmo Raft surge da necessidade de criar um algoritmo mais simples e fácil de conceber do que o Paxos, mantendo, no entanto, a eficácia do algoritmo e produzindo resultados equivalentes.

II. Raft

Como base do seu funcionamento o Raft tem o princípio de que em qualquer instante existe no máximo um servidor da rede de computação distribuída - também denominado de *cluster* - considerado líder. De forma a manter esta liderança, o líder envia periodicamente mensagens denominadas *heartbeats* sinalizando que o líder se mantém em funcionamento. Na ausência desta mensagem os restantes servidores irão começar um processo de eleição de líder que será descrito no capítulo II-A.

Este algoritmo assenta ainda num princípio de divisão temporal em *terms* que variam arbitrariamente em duração. Cada servidor regista qual o *term* atual podendo ser necessário que este se atualize caso receba alguma mensagem de qualquer outro servidor com um *term* mais atual. O *term* é incrementado sempre que se realiza uma eleição.

Assim, o Raft implementa um algoritmo de replicação de máquinas de estado 1 em vários servidores . Esses servidores devem manter um *log* de mensagens que deve ser idêntico em todos os servidores. Essas mensagens são introduzidas por um cliente diretamente no líder do *cluster*, que replica essa mensagem nos restantes servidores - seguidores, procedendo como descrito adiante no capítulo II-B.

A. Eleição de Líder

Tal como referido anteriormente, o algoritmo Raft baseia-se numa arquitetura com, no máximo, um líder. Esse líder envia periodicamente mensagens aos seguidores - *heartbeats*. Essa mensagem serve para indicar aos seguidores que o líder ainda está em funcionamento. Quando o líder deixa de funcionar os seguidores detetam a ausência de *heartbeats* e começam uma nova eleição.

Quando em funcionamento, um servidor encontra-se em um dos três estados representados na Figura 1 que descreve a máquina de estados replicada em todos os servidores.

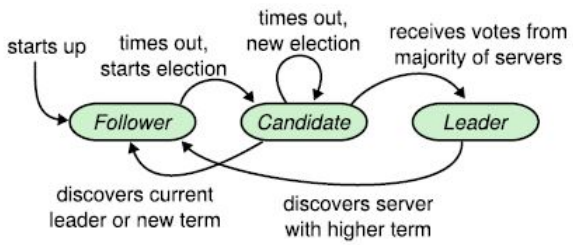

Figura 1. Maquina de estados implementada nos servidores para eleição de líder [1]

Cada servidor guarda um valor temporal aleatório, entre 150 e 350 milissegundos, que utiliza como *timeout*. Dentro desse período de tempo, se um servidor não receber uma mensagem de *heartbeat* ou uma mensagem de pedido de voto, torna-se candidato a líder.

Para iniciar a candidatura, o servidor incrementa o valor do *term*, envia uma mensagem pedindo o voto a todos os outros servidores do *cluster* e vota em si mesmo, sendo que todos os servidores apenas podem realizar um voto por *term*. De seguida, tal como representado na Figura 1, esse servidor passa a líder se receber uma maioria dos votos dos membros do *cluster*, volta a ser seguidor se outro servidor se declarar como líder, devendo para isso mostrar que tem um *term* igual ou superior ao candidato, ou mantém-se como candidato até algum servidor incorrer novamente em *timeout* e iniciar uma eleição.

O processo de eleição de líder e a garantia de existência de um líder são importantes no contexto de replicação de *logs* explicado de seguida.

B. Replicação de logs

Quando um líder é eleito este deve receber comandos do cliente e replica-los nos restantes servidores. Quando o líder recebe uma mensagem do cliente este envia em paralelo um comando para os restantes seguidores indicando que estes devem guardar essa mensagem no seu *log* - onde são armazenadas todas as mensagens. Uma mensagem define-se como *comitted* quando foi corretamente replicada pela maioria dos servidores do *cluster*.

Cada mensagem tem um índice associado que funciona como identificador da mensagem. O líder mantém registado para todos os seguidores qual o índice da próxima mensagem que será enviada aos seguidores e nunca elimina entradas do seu próprio *log*. Assim, o algoritmo de replicação de *logs* do Raft permite garantir duas propriedades:

1) Se duas mensagens em dois *logs* diferentes tiverem índices e *term* iguais então elas contêm o mesmo comando.
2) Se duas mensagens em dois *logs* diferentes tiverem índices e *term* iguais então todas as mensagens que as antecedem são idênticas.

Isto é conseguido sabendo que o líder não envia mais que uma vez uma mensagem com o mesmo índice para os seus seguidores no mesmo *term*, sendo que é importante relembrar que quando um novo líder é eleito o *term* é incrementado. E também devido à maneira como Raft lida com inconsistências nos *logs*: sempre que o líder ordena um seguidor a guardar uma mensagem este verifica também se a mensagem anteriormente guardada nesse seguidor está correta. Se não estiver coerente com a sua, o líder apaga entradas no *log* do seguidor até que que finalmente ambas as entradas sejam iguais e volta então a introduzir as mensagens no seguidor até este estar atualizado com o *log* do líder.

III. Implementação

O algoritmo Raft foi implementado em *Java* onde um número arbitrário de *threads* executa em paralelo, cada uma emulando o funcionamento de um servidor. Os processos comunicam segundo um protocolo *UDP* numa arquitetura cliente-servidor, em que o cliente é quem introduz as mensagens no sistema e os servidores são os restantes líder e seguidores.

O funcionamento desta implementação de Raft é aproximadamente igual ao descrito no capítulo II sendo que será melhor detalhado como essa implementação foi conseguida e mencionadas algumas instâncias do código que consideramos mais relevantes.

Para uma interpretação gráfica e mais intuitiva dos resultados produzidos em cada *thread* implementou-se também uma classe - GUI, que produz uma interface gráfica como a representada na Figura 2 indicando valores referentes a esse servidor como: ID, estado, ID do servidor que ele considera ser o líder, *term* e uma indicação se o servidor se encontra em funcionamento, sendo possível interromper o funcionamento através do botão *Pause*, servindo esta funcionalidade para manualmente forçar uma eleição ou interromper um funcionamento de um servidor para verificar o comportamento do sistema.

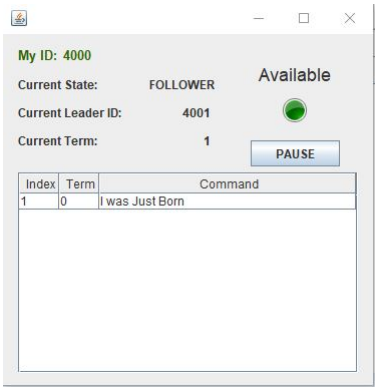

Figura 2. Interface gráfica correspondente a um servidor

A. Eleição de líder

A implementação de eleição de líder executa de maneira a que seja cumprido exatamente o funcionamento ilustrado na Figura 1. Isto é conseguido quase na sua totalidade através da classe *RaftMember* cujo diagrama UML de classes se encontra na Figura 3.

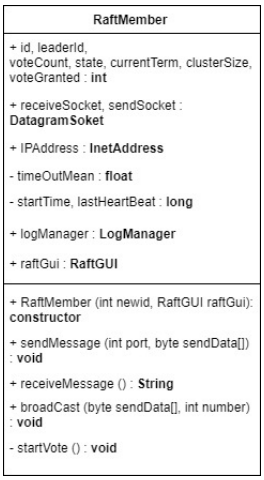

Figura 3. Diagrama UML da classe *RaftMember*

Esta classe executa numa *thread* e permite o funcionamento como candidato, seguidor ou líder. No caso de ser um seguidor responde corretamente a mensagens do tipo *heartbeat* tendo também implementado um *timeout* no *socket* UDP que faz com que o seguidor se torne um candidato na altura adequada.

Quando funciona como candidato utiliza a classe *randomGenerator* para garantir que o tempo que espera até iniciar uma nova eleição é aleatório. Quando recebe uma mensagem que indica a existência de um líder legitimo volta ao estado de seguidor e quando recebe uma maioria de votos dos restantes membros do *cluster* transita corretamente para o estado de líder.

Quando em funcionamento no estado de líder envia periodicamente mensagens de *heartbeats* aos restantes servidores e passa para o estado de seguidor quando verifica a existência de um *term* maior.

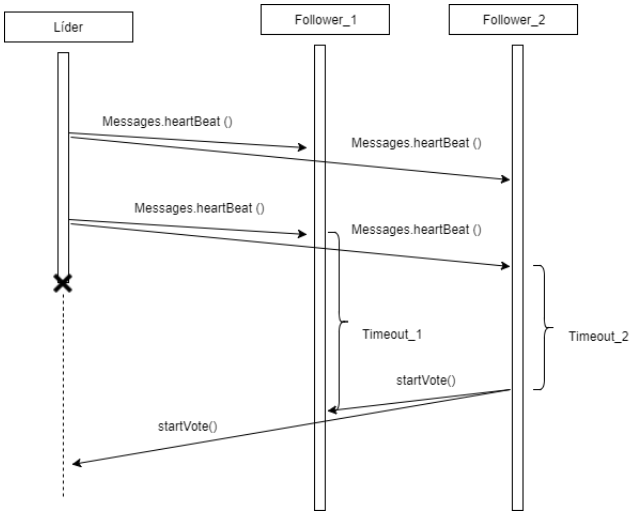

Figura 4. Funcionamento do sistema na presença de uma falha do líder

Na figura 4 encontra-se esquematizado o comportamento do sistema no caso de falha do líder. Como se pode observar, o líder envia periodicamente *hearbeats* até o momento em que falha. A partir do momento em que receberam o último *heartbeat* os seguidores iniciam um *timeout* de duração aleatória, entre 150 e 350 ms. Assim que o *timeout* ocorrer, esse seguidor inicia a candidatura enviando a mensagem *startVote*.

De maneira a introduzir falhas no lider existe um parâmetro configurável onde se pode alterar a percentagem de erros que ocorrem nas comunicações, esta percentagem vai fazer com que mensagens de *hearbeats* não sejam enviadas, o que por sua vez causa um *timeout* nos seguidores e provoca o inicio de um novo processo de eleição de lider.

B. Replicação de logs

Todos os servidores têm a si associada uma classe encarregue de gerir os seus *logs* pelo que a maioria das funcionalidades aqui descritas são realizadas por métodos desta classe, representada na Figura 5.

Quando um servidor chega à posição de líder do *cluster* começa por assumir que todos os seguidores têm os *logs* atualizados. Esta suposição não é necessariamente válida, no entanto não causa incoerências na replicação de *logs*.

Para adicionar um novo comando aos servidores, o cliente envia o novo comando para todos os servidores existentes no *cluster*. Desses servidores, apenas o líder analisa a nova entrada e segue o algoritmo no sentido de replicar a mensagem pelos *logs* dos restantes servidores de forma coerente.

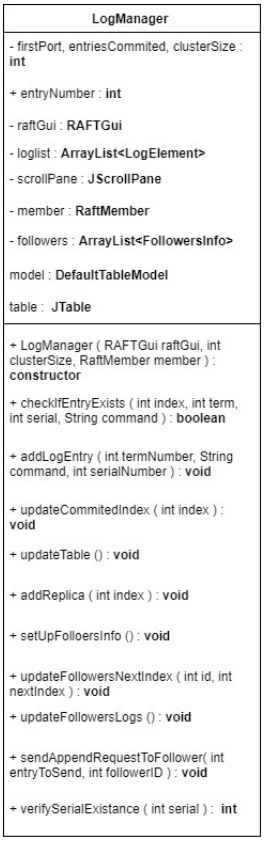

Figura 5. Diagrama UML da classe *LogManager*

Assim que recebe um novo comando dum cliente, o líder envia um pedido aos seus seguidores no sentido destes anexarem aos seus *logs* a nova entrada e todos os paramentos que lhe estão associados - numero de entrada, termo, comando. A mensagem enviada neste pedido contem o *id* do líder, o termo atual, todos os parâmetros da nova entrada e todos os parâmetros da entrada que precede esta. O *id* do líder e o *term* atual são enviados para que o seguidor saiba que está a receber uma mensagem atualizada e válida. Os parâmetros da entrada do *log* que precede a entrada que o líder pretende replicar são enviados para manter a coerência nos *logs*. Um seguidores só aceita adicionar uma nova entrada aos seus *logs* caso a sua entrada anterior esteja completamente de acordo com a entrada anterior que existe no *log* do líder.

Para replicar os *logs* de forma coerente o líder mantém para cada um dos seus seguidores um índice que corresponde à próxima entrada que deverá ser adicionada em cada um dos *logs* correspondentes.

Para cada seguidor, se o índice da próxima entrada a adicionar corresponder ao índice da próxima entrada a adicionar no líder significa que o seguidor tem o *log* atualizado e o líder não faz nada respetivamente a esse seguidor. Caso contrário, o líder tenta adicionar a nova entrada que acha que o seguidor deve receber. Se o pedido de anexo for bem sucedido o líder incrementa o índice da próxima entrada a adicionar ao seguidor. Se o pedido de anexo for rejeitado,

o líder decrementa o índice da próxima entrada a adicionar ao seguidor e tenta adicionar aquela que é a entrada que o líder considera que deve agora ser adicionada. Caso não obtenha nenhum tipo de resposta do seguidor o líder tenta continuamente enviar até que obtenha uma resposta.

Quando um cliente adiciona uma nova entrada aos servidores, essa entrada só é considerada bem sucedida se for replicada por uma maioria de servidores, só assim é que existe a garantia de que a nova entrada não pode vir a ser removida dos *logs*. Uma nova entrada é adicionada aos *logs* do líder independentemente de ser replicado por uma maioria ou não. No entanto, só é comunicado ao cliente que foi bem sucedida quando a entrada é replicada por uma maioria de servidores. Se essa replicação demorar muito tempo, o cliente pode incorrer em *timeout* e considerar que a entrada não foi adicionada com sucesso quando na verdade foi. Esta situação poderia dar origem a entradas duplicadas. O problema em questão é ultrapassado associando um *serial number* a cada entrada. Caso o cliente considere que o seu pedido falhou, o *serial number* associado aos seus comandos não é incrementado. Assim, quando o cliente tentar enviar um novo comando, este vai ter associado a si um *serial number* que já existe nos *logs*, o líder verifica que o *serial number* já existe e retorna imediatamente uma mensagem de sucesso sem adicionar de novo o comando.

De forma a manter consistência nos *logs* é necessário acrescentar uma condição na eleição de líder. Um servidor só vota num determinado candidato se esse candidato tiver os *logs* mais ou tão atualizados como ele. Desta forma, como um servidor para ser eleito líder precisa de uma maioria dos votos e as entradas dos *logs* que foram *committed* foram replicados por uma maioria, um servidor para ser eleito líder tem obrigatoriamente que conter nos seus *logs* todas as entradas que foram *committed* e portanto, essas nunca são perdidas.

IV. Resultados

Implementado o algoritmo, achou-se necessário caracterizar o seu desempenho. Seguindo o exemplo do artigo baseado na realização deste trabalho [1], dois aspetos principais foram considerados:

1) Como varia o tempo de eleição de um novo líder em função do numero de nós.
2) Comportamento do *cluster* variando a percentagem de erros nas comunicações.

Para avaliar o primeiro ponto introduziu-se 5% de erros nas comunicações realizadas pelos nós do *cluster*. De seguida registou-se o intervalo de tempo entre o momento em que um nó sofre um *timeout*, devido a uma falha nas mensagens de *heartbeat*, e o momento em que um novo nó é eleito líder do *cluster*. Utilizaram-se então esses dados para produzir a função de probabilidade acumulada representada na Figura 6.

Esta simulação foi executada para diferentes números de nós no *cluster* como se pode observar na Figura 6. É imediatamente possível concluir que todos os casos convergem, atingindo eventualmente uma probabilidade de 100% de ter sido eleito um líder nesse tempo. Conclui-se também que quanto maior é o número de nós do *cluster* mais lenta é essa convergência, o que se deve ao facto de existirem mais mensagens a serem trocadas com o aumento do número de nós.

Podemos então observar, por exemplo, seguindo a linha azul do gráfico, correspondente a 11 nós no *cluster*, que em $1ms$ é possível atribuir uma probabilidade de aproximadamente 60% de já ter sido eleito um novo líder e para $2ms$ existe uma probabilidade de aproximadamente 90% de um líder já ter sido estabelecido.

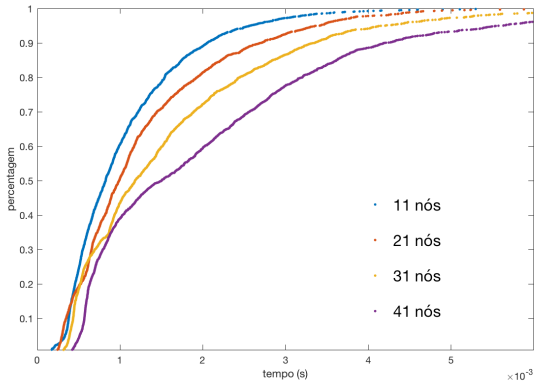

Figura 6. Tempo de eleição de líder variando o numero de nós do *cluster*

Com o intuito agora de avaliar o segundo *item* relativo ao comportamento do *cluster* em função do numero de erros nas comunicações, efetuou-se a medição do intervalo de tempo decorrido entre duas falhas consecutivas do líder. Este teste foi realizado para um numero fixo de nós, neste caso 11 nós. Visto que esta implementação executa num único computador e as comunicações são extremamente fiáveis nestas condições, foi necessário implementar a probabilidade de falha de transmissão através de uma variável com valor aleatório. Estas medições encontram-se representadas novamente na função de probabilidade acumulada da Figura 7.

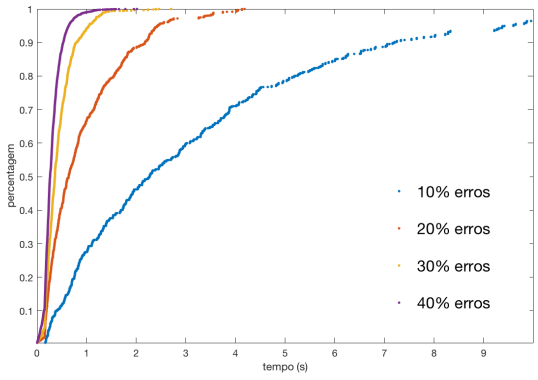

Figura 7. Tempo entre duas falhas consecutivas variando a probabilidade de erro de transmissão de uma mensagem

Podemos observar no gráfico resultante aquilo que era esperado, o tempo entre falhas consecutivas, *mean time between*

failure, decrementa com o aumento da percentagem de erros nas comunicações. Para uma percentagem de erros nas comunicações de até 10% o algoritmo consegue trabalhar de forma eficiente, no entanto, para valores mais altos o algoritmo está constantemente a necessitar de eleger um novo líder e deixa de ser viável como algoritmo de replicação de *logs*. Embora continue coerente na replicação de *logs* consideramos que a *availability* do algoritmo passa para valores não aceitáveis. Seguindo a linha vermelha, por exemplo, observa-se que em aproximadamente 70% dos casos o tempo entre duas falhas foi de $1ms$ ou menos.

Julgou-se ainda pertinente caracterizar o algoritmo na sua eficiência na replicação de *logs*. Como tal, efetuaram-se medições do intervalo de tempo ocorrido entre o instante em que o líder recebeu uma nova mensagem do cliente até o instante em que se completou a replicação da mensagem numa maioria dos nós. Efetuaram-se novamente estas medições variando a probabilidade de erro na transmissão de uma mensagem, como se pode observar na Figura 8 e mantendo o *cluster* constantemente com 11 nós.

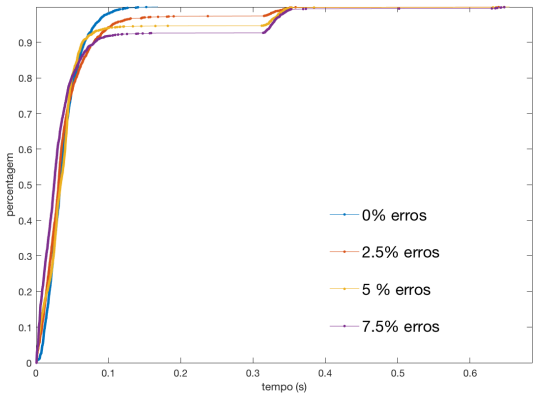

Figura 8. Tempo necessário para replicar uma mensagem pela maioria dos nós do *cluster* variando a probabilidade de erro na transmissão de mensagens

Observa-se então que, no caso de não existirem falhas na transmissão de mensagens (linha azul), todos os casos tiveram um tempo de replicação dos *logs* de menos de $150ms$, que corresponde simplesmente ao tempo de execução de todas as funções envolvidas no processo de receção da mensagem do cliente, e replicação da nova entrada nos *logs* de uma maioria dos elementos do *cluster*.

Nas restantes curvas apresentadas, com erros de $2,5\%$, 5% e $7,5\%$ observa-se um comportamento análogo entre elas. Tomando como exemplo a curva a amarelo relativa a 5% de falhas, observa-se que em cerca de 95% dos casos os *logs* são replicados com sucesso num tempo menor ou igual a $150ms$. No entanto, observa-se que os restantes 5% são completados em torno dos $300ms$. Isto deve-se ao facto de a tentativa do líder replicar os *logs* pelos seguidores ter falhado. Quando isto acontece o cliente assume que a nova entrada não foi adicionada aos *logs* e tenta enviar de novo. O *timeout* do cliente é de $300ms$ daí 5% das tentativas demorarem mais de $300ms$, a tentativa de replicar falha 5% das vezes e o reenvio ocorre $300ms$ depois da primeira tentativa.

Por fim, analisou-se ainda a carga do CPU em percentagem face ao número de nós pertencentes ao *cluster* no gráfico de probabilidade acumulada na Figura 9. (Teste realizado com 5% de erros nas comunicações)

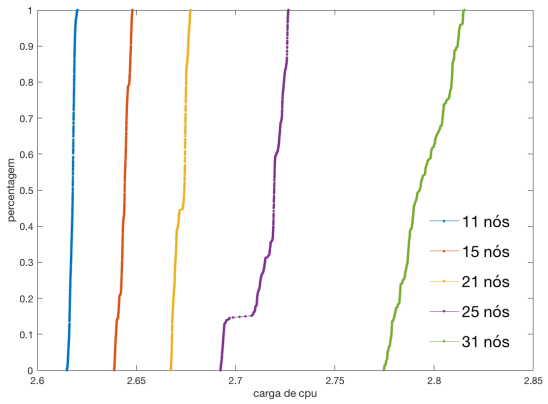

Figura 9. Carga do CPU variando o numero de nós

Dois aspetos deviam ser referidos em relação à Figura 9:

1) Como seria de esperar, um aumento no número de nós no *cluster* traduz-se numa maior percentagem de utilização do CPU pois existem mais processos a executar paralelamente.
2) Verifica-se ainda que com o aumento do número de nós existe uma maior variação da utilização do CPU que se deve tanto ao facto de existir uma variação de utilização do CPU associada a cada *thread* como ao facto de com um maior número de nós existirem falhas mais frequentemente.

No entanto em todas as curvas se verifica que a utilização do CPU se mantém aproximadamente constante visto que as curvas se aproximam de retas verticais.

V. Conclusão

Com este trabalho foi possível compreender mais a fundo este recente algoritmo de consenso que visa conseguir os mesmos resultados que *Paxos*. Verificamos que o desenvolvimento de um algoritmo cujo objetivo principal era torna-lo compreensível refletiu-se num algoritmo robusto, rápido e que oferece a possibilidade de expansão por parte de outros [1]. Infelizmente não foi possível comparar os resultados com outros algoritmos possíveis uma vez que mais nenhum grupo implementou algoritmos de consenso e também não existem muitas caracterizações disponíveis. Contudo, os resultados estão todos de acordo com o esperado e podem ser usados como referência em trabalhos futuros.

No desenvolvimento deste trabalho a distribuição de tarefas foi aproximadamente igual por todos os elementos do grupo. O Diogo Santos esteve mais envolvido na implementação de eleição de líder, a parte de replicação de *logs*, interfaces

gráficas e medições foram feitas por Nuno Fernandes e João Magalhães. O relatório final foi feito em conjunto por parte de todos os elementos. Diogo Santos 33%, Nuno Fernandes 33%, e João Magalhães 33%.

REFERÊNCIAS

[1] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," *USENIX Annual Technical Conference*, 2014.

[2] "Raft visualization." https://raft.github.io/.

[3] "Paxos made simple." https://lamport.azurewebsites.net/pubs/paxos-simple.pdf.

[4] C. Y. Qing Li, *Real-time concepts for embedded systems*. CMP Books, 2003.

[5] "Repositório *Github* do projeto." https://github.com/joao4597/RAFT.

[6] "Repositório *Github* do projeto - interface gráfica e cliente." https://github.com/joao4597/raftClient.