Mestrado em Engenharia Informática
Tolerância a Faltas Distribuídas
Relatório - 3ª Fase

Diogo Novo – 60400
João Arcanjo – 60405
João Lopes – 60493

Grupo - 4

Introdução

Com a evolução das tecnologias existe cada vez mais uma necessidade de se realizar processamento elevado, ultrapassando este muitas das vezes, a capacidade de uma única máquina. Criou-se então o conceito de sistema distribuído, que se baseia na existência de várias máquinas sincronizadas (ou não) a funcionar em paralelo para atingir um objetivo cujo seria impossível através de uma única instância, seja este obter um consenso entre todas sobre algum valor, ou computar algum resultado demasiado complexo. Obviamente estas máquinas também podem ter falhas e problemas, cabendo ao desenvolvedor do sistema saber como contorná-las e tornar o sistema todo mais robusto e tolerante.

Foi então proposto aos alunos da UC Tolerância de Faltas Distribuídas que desenvolvessem um sistema que fosse capaz de implementar o algoritmo de *consensus* RAFT, sendo este relatório alusivo a todas as três fases deste projeto.

Na primeira fase elabora-se sobre o desenvolvimento de toda a arquitetura e como foi feita toda a comunicação entre as máquinas. Já na segunda fase concentrou-se no começo do desenvolvimento do algoritmo RAFT, sendo implementado a etapa onde é feita a eleição do líder, ou seja, onde se elege autonomamente a réplica que irá comunicar com o cliente e replicar todas as informações recebidas para as restantes, sendo esta etapa também tolerante a falhas, elegendo um novo líder caso o atual falhe. Por fim, na última e terceira fase, é explicado como foi implementado todo o processo de *log replication* que se dá entre as máquinas que estão a correr o algoritmo, com o intuito de replicar e preservar a informação de forma consistente, incluindo também a explicação de como se implementou um cliente com o intuito de se testar todo o bom funcionamento da arquitetura.

1ª Fase – Communication Abstractions

Para esta primeira fase do projeto foi requisitado que se implementasse a comunicação entre as réplicas, com o intuito de todas conseguirem realizar pedidos entre elas e obter os devidos resultados das réplicas que se encontram ativa.

Com o intuito de simplificar a quantidade de código necessário para estabelecer os canais de comunicação entre as réplicas recorreu-se à *framework* gRCP, sendo esta abordagem mais direta e simples quando comparada com outras abordagens como a utilização de *sockets*. Através desta abordagem foi possível aprofundar mais o desenvolvimento de um bom funcionamento entre réplicas do que propriamente na manutenção das ligações das mesmas ao garantir que todas as comunicações através de *sockets* estariam a funcionar corretamente.

Com isto, foi definido um contrato inicial onde estão definidos os métodos existentes e o tipo de objetos que são enviados e recebidos por cada réplica. Este pode ser observado na Figura 1.

```
// The gRPC server service definition.
service Server {
  // Invoke operation
  rpc invoke(Request) returns (Result);
}

message Request {
  int32 id = 1;
  string label = 2;
  string data = 3;
  google.protobuf.Timestamp timestamp = 4;
}

message Result {
  oneof result {
    string resultMessage = 1;
    ResultList results = 2;
  }
  int32 id = 3;
  google.protobuf.Timestamp timestamp = 4;
}

message ResultList {
  repeated string list = 1;
}
```

Figura 1- Primeiro contrato definido para utilização do gRPC

Este contrato define o objeto que é enviado num pedido e também o tipo que é retornado numa resposta. Para além disso, também define o método *invoke*, sendo este o método chamado para interagir com outra réplica. É importante salientar que esta interação é independente do tipo de ação que se deseje realizar.

Para iniciar a biblioteca construída, é necessário passar por argumento o ID da réplica que está a ser inicializada e a diretoria do ficheiro que possui o <ip>:<port> tanto da réplica que está a ser inicializada como das restantes com as quais esta irá comunicar.

Através dos dados presentes no ficheiro são então criados os canais de comunicação entre a réplica atual e as restantes através da *framework* gRPC tal como referido anteriormente.

Após os canais de comunicação serem criados, todas as réplicas vão poder assumir tanto o papel de servidor como o papel de cliente, ou seja, estas vão estar habilitadas tanto para invocar operações nas restantes réplicas como serem invocadas para resolver operações pelas mesmas.

Criou-se então a classe GRPCServer que possui a componente servidora da réplica, que possuirá uma *thread* própria denominada por serverThread, em que esta estará a processar os pedidos provenientes das restantes. Após a componente servidora estar iniciada, é criada pela *thread* principal uma outra *thread*, denominada por resultsThread, que terá como papel interpretar os resultados que irão chegar à *blockingQueue*. Esta *thread* foi criada com o objetivo de não bloquear o processo principal quando se está à espera de algum resultado, visto que, apesar de não ser possível processar nenhum novo pedido (ADD ou GET) quando nos encontramos à espera da resposta a outro anteriormente realizado, considerámos que caso pretendamos terminar a execução da atual réplica, não devemos estar dependentes da resposta de terceiros que pode nunca chegar. A Figura 2 representa todas as *threads* criadas bem como os diferentes componentes se relacionam.

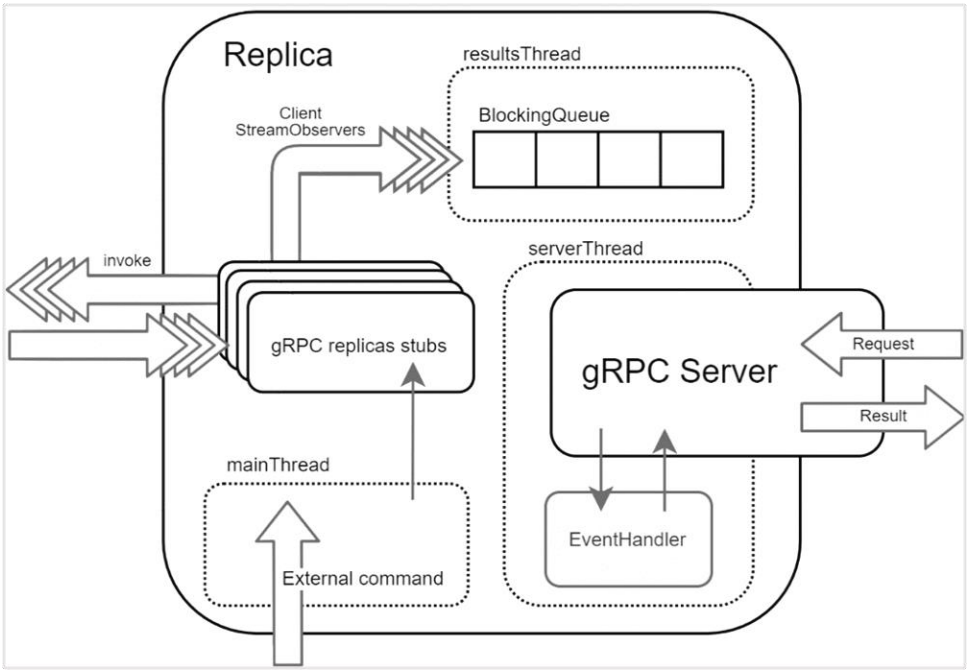

Figura 2- Arquitetura da solução desenvolvida

Com esta arquitetura foi então possível desenvolver o algoritmo pretendido, enviar uma mensagem que atingisse todas as outras réplicas (ADD), fazendo com que estas invocassem a ação desejada. Para garantir alguma consistência também se garantiu que apenas seria possível realizar um novo pedido caso a máquina que fez o pedido recebesse as respostas de pelo menos metade do número de réplicas no sistema.

Embora útil, esta função acaba por gerar outro problema, nomeadamente caso uma resposta chegue atrasada, ou seja, já depois da máquina que desencadeou o pedido ter recebido mais de metade das respostas e ter sido desencadeado um novo pedido. Como forma de contornar isto, adicionou-se aos pedidos uma *timestamp*, que diz respeito ao instante em que foi desencadeado o pedido, assim, torna-se possível garantir que mesmo que a resposta chegue atrasada, ao comparar a sua *timestamp* com a última registada na máquina que enviou o pedido, caso estas não sejam idênticas descarta-se a resposta, visto que já não tem importância.

2ª Fase – Leader election

Como segunda fase do projeto foi requisitado que se iniciasse a implementação da eleição do líder entre as réplicas e a alteração do mesmo no caso deste falhar.

Para isso, reutilizou-se a main *thread*, para que esta realizasse todo o processamento, ou seja, esta está encarregue de toda a lógica da eleição do líder. Quando a réplica se encontra no estado de *follower* a *thread* será responsável por esperar por *heartbeats* provenientes do líder, no caso de se encontrar no estado de *candidate*, esta *thread* será responsável por enviar os *requestVotes* para as restantes réplicas e esperar pelos votos das mesmas, e no caso da réplica se encontrar no estado de *leader*, a *thread* será responsável por enviar *heartbeats* para as restantes réplicas.

Como forma de controlar todo o comportamento, todas as réplicas quando iniciam, arrancam como *followers*, ficando à espera de receber algum *heartbeat*, fazendo uma espera passiva utilizando o método *await*. Caso esta receba alguma mensagem do líder, o *follower* é então notificado, atualiza o tempo que se irá encontrar à espera e volta a colocar-se em espera até receber outro *heartbeat*, caso o tempo de espera expire, a réplica transita para o estado de *candidate*, incrementando o seu *term* e propagando esta informação para as restantes réplicas. Após isso, a mesma fica a aguardar pela maioria dos votos, e caso durante o processo, esta receba um *heartbeat* proveniente de outra réplica que se considera líder e que possua um *term* superior, o processo é abortado e a réplica retorna ao estado *follower*, seguindo o líder que enviou o respetivo *heartbeat*.

Ênfase no facto de que todo o comportamento relativo à interpretação dos termos é executado sempre quando é recebida uma mensagem, tornando assim este comportamento independente de todo o processo de eleição do novo líder.

Para o desenvolvimento desta fase, foi necessário criarmos três classes que iram ser fundamentais para que a funcionalidade de leader election funcione corretamente pois iram possuir todos os dados que necessitamos. A descrição das mesmas apresenta-se seguidamente:

- **State**: Esta classe é instanciada no ínicio de cada réplica e tem como objetivo, tal como o nome indica, armazenar o estado atual da mesma. Através desta classe é possível obter qual o "papel" da réplica, o *term* em que a mesma se encontra, em quem é que a réplica votou no atual *term* e a lista de *logs*.
- **RequestVoteRPC**: Esta classe por sua vez possui duas *data* classes, *RequestVoteArgs* e *ResultVote*, que iram conter os dados enviados por um determinado candidato para as restantes réplicas e as respostas dessas réplicas para os candidatos respetivamente.
- **AppendEntriesRPC**: Esta classe possui uma data classe à qual denominamos por AppendEntriesArgs que contém os dados provenientes no *heartbeat* enviados do líder para todas as restantes réplicas.

Na figura 3 apresentamos um esquema que ilustra a comunicação entre as réplicas no processo de escolha do líder, realçando a alteração dos estados das mesmas ao longo do tempo.

Todas as cinco réplicas começam no estado *follower*, no qual, como referido acima, ficam à espera de *heartbeats* provenientes do líder durante um período variável de réplica para réplica. Como não existe qualquer líder eleito no início da execução do algoritmo, a réplica que possui um tempo de espera por *heartbeat* inferior, vai ser a primeira a transitar para o estado de candidato. No passo por (1) representado na figura, podemos verificar este acontecimento, no qual a réplica 2 transitou para o estado de candidato. No passo (2) a mesma réplica realiza um pedido de votos a todas as restantes, e quando recebe os votos da maioria (incluindo o seu auto-voto) vai transitar para o estado de *leader* tal como se pode verificar no passo (3). Neste estado, a réplica vai enviando *hearbeats* para as restantes réplicas (4) e (5). No passo (6) a réplica 1 acabou por dar *timeout* pois não recebeu nenhum *heartbeat* no intervalo de tempo estipulado por si e consequentemente esta transita para o estado *candidate*, incrementa o seu *term* que até à altura era igual ao *term* do líder, e envia pedidos de votos para todas as réplicas. Estas ao verificarem que o candidato possui um *term* superior ao delas, atualizam o seu *term* e enviam o seu voto para o candidato, inclusivamente o atual líder pois possuí um term inferior ao do candidato, transitando consequentemente para o estado de *follower* tal comoé verificável no passo (8). No passo (9) a réplica

candidata, ao receber os votos necessários, transita para o estado *leader*. Nos passos (10) e (11) podemos verificar os *heartbeats* enviados, agora da réplica 1 para as restantes.

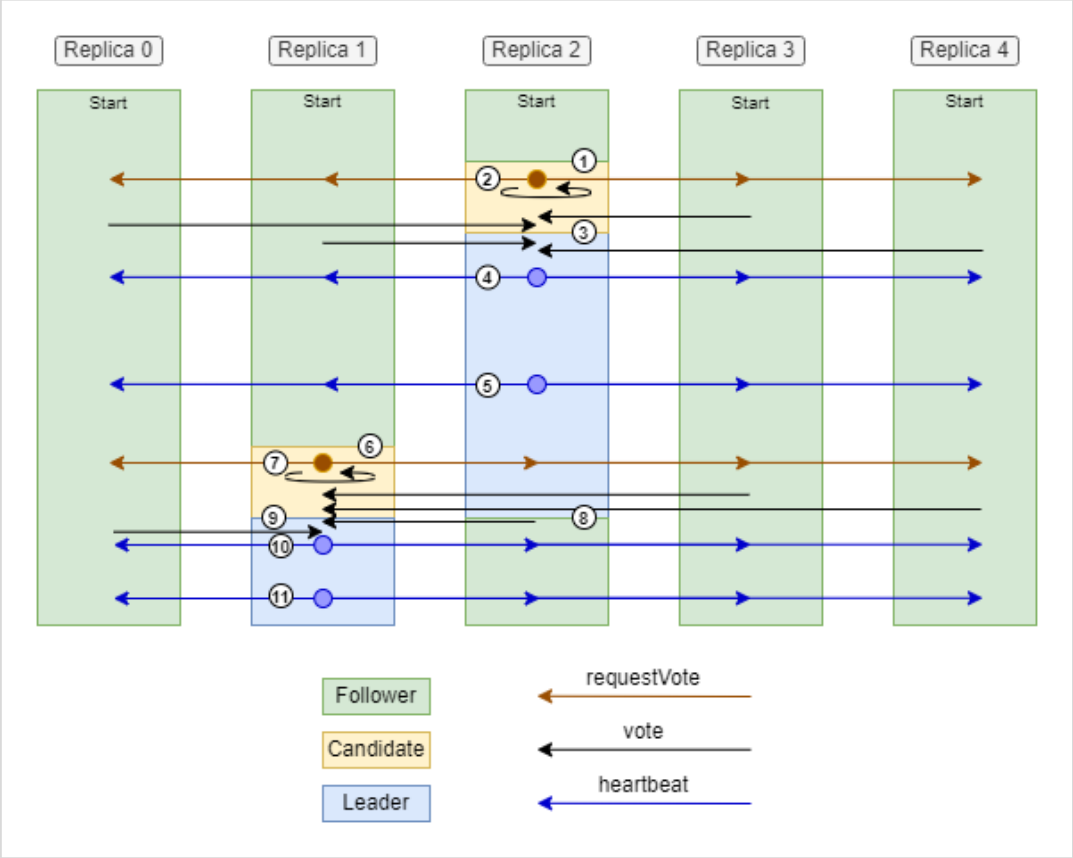

Figura 3- Simulação da comunicação durante o período de eleição de líder.

3ª Fase – Log Replication

Nesta fase foi realizado um elevado número de alterações, com o intuito de que agora, um ou mais clientes possam realizar o envio de um determinado comando para a réplica líder, sendo estes comandos depois replicados para todas as restantes réplicas, de maneira a que todas possuam a sua máquina de estados com o mesmo estado que a do líder. No nosso sistema, os comandos são designados por *elements* e que se encontram no *log* de cada réplica, sendo os *logs committed* também armazenados num ficheiro.

3.1 – Desenvolvimento do cliente

Começou-se então por desenvolver-se o cliente. Tendo em conta que este iria comunicar diretamente com as réplicas, foi desenvolvida toda a comunicação utilizando também gRPC. Para tal, foi então alterado o contrato previamente já mencionado para que este agora contivesse o novo método *request*. Este por sua vez, irá ser chamado pelo cliente, ao qual irá passar por parâmetro um objeto que contém tanto o tipo de ação pretendida, bem como o *payload* da mesma.

O comportamento do cliente, foi definido para que ao ser iniciado, se dirija a uma réplica, visto conhecer os endereços de todas através do ficheiro de configuração também utilizado quando se inicia as réplicas. Após estabelecer a ligação com uma das réplicas, existem duas possibilidades, esta ser o líder e a comunicação prosseguir como pretendido, ou a réplica que foi comunicada, encontra-se noutro

estado e não ser capaz de satisfazer os pedidos, indicando, caso exista, o líder atual para que o cliente saiba com quem comunicar.

Para isso foi necessário, assim que o cliente inicia, inicializar todos os canais de comunicação com todas as réplicas do sistema. Estes canais, são *stubs* bloqueantes devido à natureza das respostas do sistema, sendo estas apenas uma mensagem meramente informativa.

Após todos os passos referidos, o cliente limita-se a gerar valores aleatórios dentro de um certo intervalo (1 a 5), enviando-os para a réplica líder. Caso em algum instante o cliente receba a informação de que existe um novo líder, o mesmo apenas troca de canais, reenviando a mensagem que não foi bem-sucedida.

3.2 – Alterações realizadas

Como já mencionado, o contrato gRPC sofreu algumas alterações, este agora inclui um novo método *request* para receber os pedidos do cliente, que, ao serem recebidos pelo líder, serão armazenados numa nova *queue*. Caso seja recebido um pedido por uma réplica que não é o líder, a mesma apenas responde com o id do líder para qual o cliente tem que redirecionar os seus pedidos. Os resultados inseridos nesta *queue* são então interpretados por uma nova *thread* que está apenas responsável por esta ação, ou seja, ler os pedidos do cliente, validar os mesmos, e enviar para todos os *followers* o pedido bem, como outras *entries* que se encontrarem no seu *log* e que ainda não tenham sido enviadas. A arquitetura da solução final, pode ser observada na figura 4.

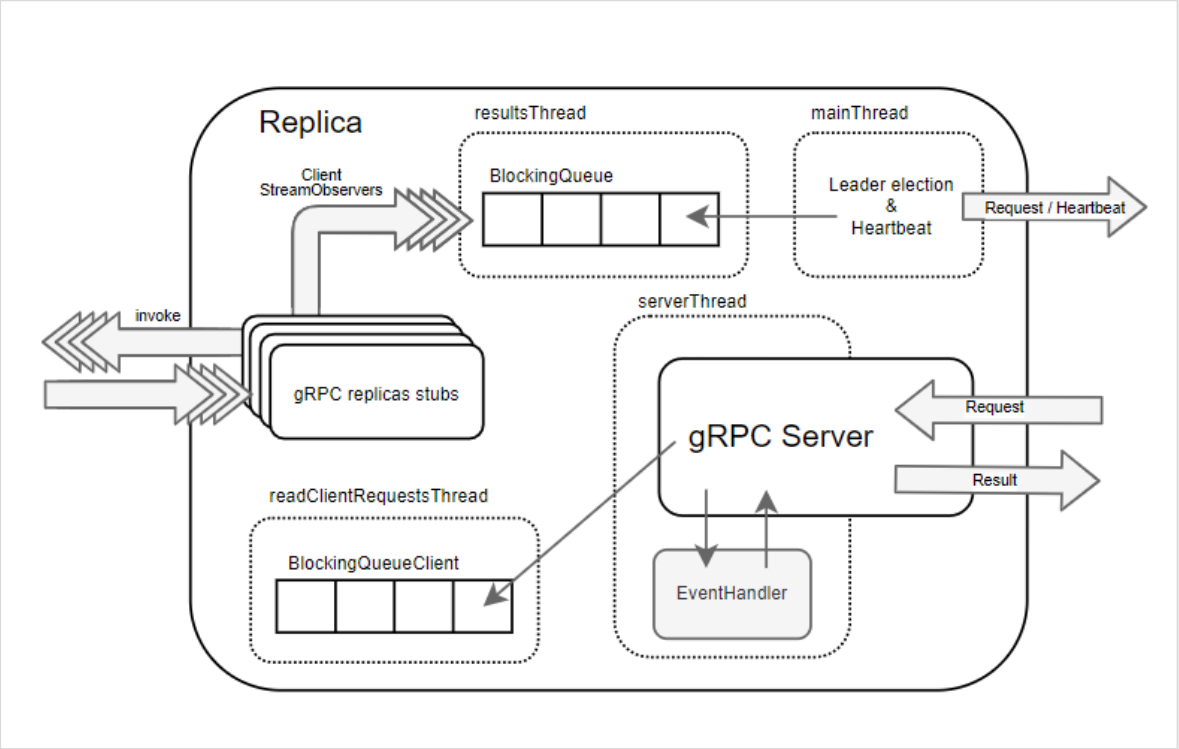

Figura 4 – Arquitetura final da solução desenvolvida

Relativamente ao contrato gRPC, para além deste conter um novo método, o mesmo sofreu também alterações no tipo *Request*, bem como no tipo *Result*. As alterações foram realizadas sobre campos, cujo tipo eram *string*, tendo sido alterados para o tipo *bytes*, com o intuito de tornar todo o processo mais genérico. O contrato final pode ser observado na Figura 5.

```
// The gRPC server service definition.
service Server {
  // Invoke operation
  rpc invoke(Request) returns (Result);
  rpc request(Request) returns (Result);
}

message Request {
  int32 id = 1;
  string label = 2;
  bytes data = 3;
  google.protobuf.Timestamp timestamp = 4;
}

message Result {
  oneof result {
    string resultMessage = 1;
    bytes results = 2;

  }
  int32 id = 3;
  google.protobuf.Timestamp timestamp = 4;
  string label = 5;
}
```

Figura 5 – Contrato final do protobuf

De forma a complementar toda a implementação, relativamente à fase anterior, foi adicionada complexidade à classe *State*. Nesta, foi adicionada uma lista que representa o *log* interno da máquina, uma referência para um ficheiro que possui os *logs* que foram *committed*.

Outros campos também adicionados à classe *State*, foram nomeadamente *commitIndex*, *lastApplied*, *lastLogIndex*, *lastLogTerm*, *nextPairToFile*, *nextIndex* e *stateMachine*. Devido à sua importância para a lógica por trás desta fase, a utilidade de cada um será mencionada de seguida.

- **commitIndex:** Este campo tem como objetivo principal, armazenar em cada réplica qual a próxima *entry* que pode ser *committed*, ou seja, a operação seguinte presente no *log* que pode afetar a máquina de estados.
- **lastApplied:** Possui o último *index* que a respetiva réplica fez *commit*. Este nunca é superior ao campo *commitIndex* e é atualizado quando é feita uma afetação (*commit*) na máquina de estados.
- **lastLogIndex:** Contem o maior *index* do *log*, ou seja, o *index* da última *entry* que foi recebida pelo líder.
- **lastLogTerm:** Este campo possui o *term* da última *entry* recebida pelo líder, sendo crucial para verificar se futuras *entries* enviadas por parte do líder são válidas e também para garantir que não são eleitos líderes com termos inferiores aos já armazenados no *log*.
- **nextPairToFile:** Este campo corresponde ao intervalo de *indexes* que serão escritos no ficheiro de *logs* (processo descrito em seguida). O objetivo deste campo é permitir que não se escreva no ficheiro cada vez que uma *entry* é *committed*, sendo assim possível escrever no ficheiro um intervalo de *entries* de cada vez (por exemplo da posição 1 à 10).
- **nextIndex:** Este campo apenas é utilizado pelo líder e representa uma lista que possuí o próximo *index* que cada uma das restantes réplicas pretende receber. Esta lista é atualizada quando uma réplica responde ao *líder* após receber uma determinada *entry* (que não seja *heartbeat)* por parte deste, permitindo assim o envio de informação conforme a necessidade de cada réplica.

- **stateMachine:** Este campo representa nada mais que a instância da máquina de estados (classe StateMachine) da respetiva réplica.

A *classe* "StateMachine" representa, como o nome indica, a máquina de estados interna da réplica, sendo esta modificada em dois momentos, inicialmente quando é configurada conforme as informações previamente armazenadas nos ficheiros e sempre que é realizado o *commit* de alguma *entry*. Além disso a mesma consegue interpretar o tipo de ação passada pela ação designada em cada entrada do *log* e reagir conforme tal.

Quanto à escrita de ficheiros, cada réplica contém agora um ficheiro **.txt** sendo este lido inicialmente quando a réplica arranca, com o intuito de recuperar eventuais *log entries* que estivessem *committed*, quando esta terminou de forma abrupta. Para além disso, como já foi mencionado, são realizadas escritas quando se tenciona salvar informação de forma permanente naquela réplica. Importante salientar que estas escritas não acontecem sempre que se dá um *commit* por questões de eficiência, tendo se achado errado, estar constantemente a manipular informações num ficheiro, tendo se optado por ter um contador que ao fim de *x commits* é que realmente faz a inserção destes no ficheiro.

Relativamente à replicação de informação, foi necessário garantir que ao receber um novo pedido do cliente, esta informação fosse replicada para as restantes réplicas através de algo semelhante ao *quoromInvoke* já implementado. A única diferença é que em vez de enviar uma única informação comum para todas as réplicas, seria necessário variar a mesma, conforme o *nextIndex* que cada uma das réplicas pretende receber. Esta informação é por sua vez convertida numa *string* em formato *json*, quando é realizado o envio, sendo novamente reconstruída do lado do *follower*, para ser interpretada.

Para além disso também foi necessário realizar algumas alterações quanto à interpretação do *heartbeat* e ao que este inclui. Para além de enviar uma lista de *entries* vazia, é incluído neste o *commitIndex* do líder. Esta adição permite que os *followers* sejam também notificados através do *heartbeat* relativamente ao próximo *index* que podem realizar *commit*, não sendo necessário um novo pedido do cliente para estes serem notificados de uma eventual atualização relativamente a este parâmetro.

Outra alteração realizada no *heartbeat*, foi a lógica de como ocorriam os *timeouts*, agora, o mesmo só expira, caso não tenha recebido nenhum *heartbeat* e não ter recebido também nenhuma *entry* (novo pedido proveniente de um cliente). Algo semelhante também acontece do lado do líder, através da ativação de uma *flag* assim que é feito o envio de *entries*, evitando assim um envio de *heartbeat* desnecessário.

3.3 – Replicação dos Logs

Quanto à lógica por trás de todo o processo, o mesmo é descrito da seguinte seguidamente. O cliente realiza um pedido a uma das réplicas, sendo sempre redirecionado para o líder. Após este fazer o envio das informações desejadas, a réplica que se encontra como líder, ao receber a informação, adiciona-a ao seu *log*, replicando-a em seguida para todas as outras, incluindo, nessa replicação, todas as *entries* que cada *follower* necessita. Os *followers* ao receberem as *entries* (caso sejam as *entries* que pretendam receber), aplicam-nas no seu *log*, e informam o líder qual é o *index* que pretendem receber na próxima vez. Quando o líder recebe as respostas dos *followers*, irá atualizar o *nextIndex* de cada *follower* e irá verificar se pode atualizar o seu *commitIndex*, ou seja, se maioria das réplicas recebeu a informação enviada. Caso tal aconteça, atualiza a máquina de estados e realiza todos os processos adjacentes, tais como, adicionar a *entry committed* ao *file*. O próximo *AppendEntry* enviado para os *followers*, para além de conter todas as *entrys* que os followers necessitam, irá também incluir o *commitIndex* do líder atualizado, fazendo com que os *followers* também realizem a atualização da sua máquina de estados.

A Figura 6 tem como objetivo exemplificar os casos em que um determinado *follower* "acordou" e os *logs* que possui no seu ficheiro, não estão atualizados quando comparados com os do líder, que, tem estado a receber os pedidos dos clientes e a enviá-los para os restantes *followers*. De realçar que quando um líder envia um *Append Entry* para um determinado *follower*, também envia para os restantes, embora na Figura 6 apenas existir um *follower* e um líder, de maneira a simplificar a explicação.

O passo (1) indica o momento em que o líder recebeu um pedido do cliente. Seguidamente, no passo (2) irá enviar esse pedido e as *entries* que ele julga ser as que o respetivo *follower* necessita para o mesmo. O *follower*, no passo (3), ao receber as *entries*, e ao verificar que o *prevLogTerm* e o *prevLogIndex* não coincidem com os valores do mesmo, o *follower* não irá atualizar o seu *log* e irá informar o líder, no passo (4), a partir de qual *index* é que quer voltar a receber as *entries*. O líder, no passo (5), ao receber a resposta, irá atualizar o *nextIndex* relativamente a este *follower*, e, visto as restantes réplicas que já estavam ativas, terem aceite as *entries* enviadas, o líder vai recalcular o *commitIndex*. Caso seja um valor novo, irá atualizar a sua máquina de estados e irá atualizar o ficheiro que possui os *logs committed*. O passo (6) representa um novo *appendEntries*, originado pelo líder devido a ter surgido um novo pedido proveniente de um cliente. Este *appendEntries*, por sua vez, irá incluir todas as *entries* que o *follower* necessita e o *commitIndex* calculado pelo líder, referido anteriormente. No passo (7) o *follower* vai atualizar o seu *log*, vai guardar o *commitIndex*, e procede à atualização da máquina de estados, atualizando também o ficheiro que possui os seus *logs committed*. No passo (8), a resposta da réplica irá conter o seu *nextIndex*, que deve estar de acordo com a maioria dos *nextIndexs* das restantes réplicas. O passo (9) é igual ao passo (5).

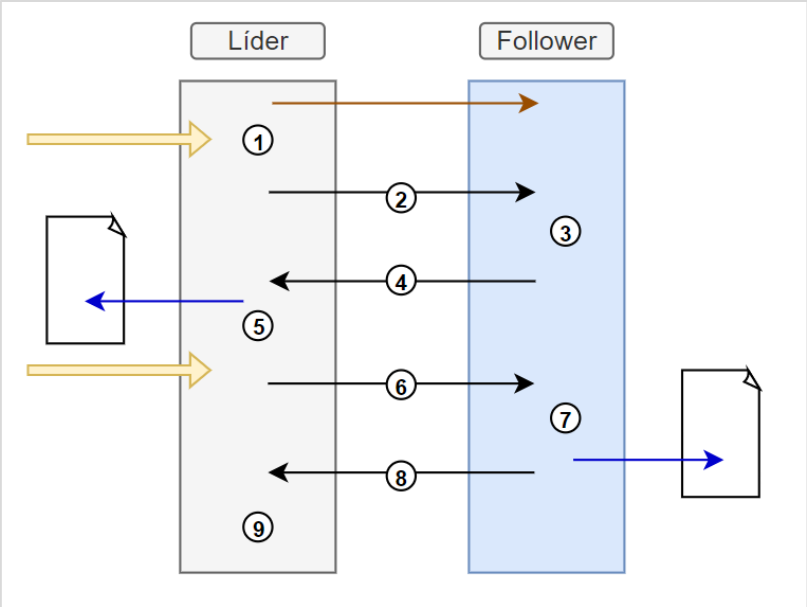

Figura 6 – Replicação de *logs* entre um *follower* e o líder

Conclusão

Finda a elaboração deste trabalho, foi-nos possível não só consolidar a matéria lecionada na UC de Tolerância de Faltas Distribuídas, como também aprender mais sobre como é possível desenvolver sistemas que comunicam entre si de forma a criar soluções complexas e modernas.

O facto de todo o processo ter evoluído através de etapas, num processo gradual e de forma natural, facilitou não só a sua compreensão como também a sua implementação, embora tal nunca tivesse sido feito por nenhum dos elementos do grupo, adquirindo assim conhecimentos mais avançados sobre um tópico bastante relevante para toda a UC.

Mesmo assim acreditamos que existe espaço para melhorar, não pela má implementação de uma solução, visto que se considera que a mesma consegue cumprir todos os objetivos, mas sim, pela grande complexidade e profundidade que existe quanto a este tópico que está em constante evolução.