INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA
Área Departamental de Engenharia Eletrónica e Telecomunicações e de Computadores

# Projeto Final de Avaliação

## Computação Distribuída

*Versão 1.4*

**Docentes:**
Prof. Doutor Luís Assunção

**Alunos:**
36868 - Bruno Costa
42113 - António F. Oliveira
44178 - Rodrigo Pina

# Histórico de Revisões

| Data | Versão | Descrição | Autor |
|---|---|---|---|
| **29/12/2020** | **1.0** | **Aspetos do config_svc.** | **G11** |
| **10/01/2021** | **1.1** | **Config_svc e protocolo mensagens.** | **G11** |
| **12/01/2021** | **1.2** | **Config_svc – DISCOVERY_REQ e DISCOVERY_ACK.** | **G11** |
| **14/01/2021** | **1.3** | **Mecanismo de consenso e coordenação.** | **G11** |
| **15/01/2021** | **1.4** | **Adição de figuras e diagramas.** | **G11** |

# Tabela de Conteúdos

# INTRODUÇÃO

Um sistema distribuído pode ser, de uma forma simples, caracterizado como uma coleção independente de máquinas/instâncias que colaboram entre si e que da perspetiva dos utilizadores são vistas como um único sistema coerente.
As formas e tecnologias usadas para a implementação dessa colaboração entre máquinas constituem o foco principal do desenvolvimento deste tipo de sistemas.

Neste documento, será descrita a implementação do projeto final da Unidade Curricular de Computação Distribuída no que diz respeito à arquitetura, responsabilidades de cada parte e aspetos de replicação, consenso, balanceamento de carga e tolerâncias a falhas.

## Objetivos

Neste seguimento, o projeto final teve os seguintes objetivos:

- Desenvolver um sistema distribuído de armazenamento de dados (Chave, Valor), usando o *middleware* gRPC (Google Remote Procedure Call) e o Spread toolkit de comunicação por grupos, num cluster de servidores com replicação de dados e um modelo de consistência garantido por consenso entre todos os servidores usando comunicação por grupos;
- Respeitar um conjunto de requisitos funcionais específicos (que constam no enunciado do projeto). No decorrer das várias explicações serão feitas menções a cada um destes.

# 1. Composição e Arquitetura do Sistema

O cluster é composto por uma ou mais instâncias de aplicações servidoras com repositórios associados e um serviço de configurações. **A comunicação entre** as várias **instâncias servidoras** é feita **via *Spread***, **num grupo denominado Clusters**. De seguida, são descritas cada uma destas partes.

## 1.1 Serviço de Configurações

O serviço de gestão de configurações tem um IP e porto bem conhecido e tem como principal função notificar os clientes do cluster sobre configuração dinâmica do cluster, isto é, quais os membros do cluster que estão em operação.

Em adição, apresenta também um papel importante de despoletar o processo de eleição de líder do grupo.

## 1.2 Aplicações Servidoras

O sistema consiste num cluster composto por várias aplicações servidoras que têm como objetivo providenciar aos clientes a persistência (escrita) e consulta (leitura) de dados.

Todas as aplicações suportam operações de leitura, no entanto apenas uma, que incorpora o papel de líder do grupo, permite efetuar a escrita de valores e autorizar a modificação de dados do cluster, nomeadamente remoção de objetos.

## 1.3 Aplicações Cliente

Por fim, as aplicações cliente consistem no meio de interação com o cluster usado pelos utilizadores. São atualizadas pelo Serviço de Configurações e disponibilizam os menus de escrita e leituras de valores.

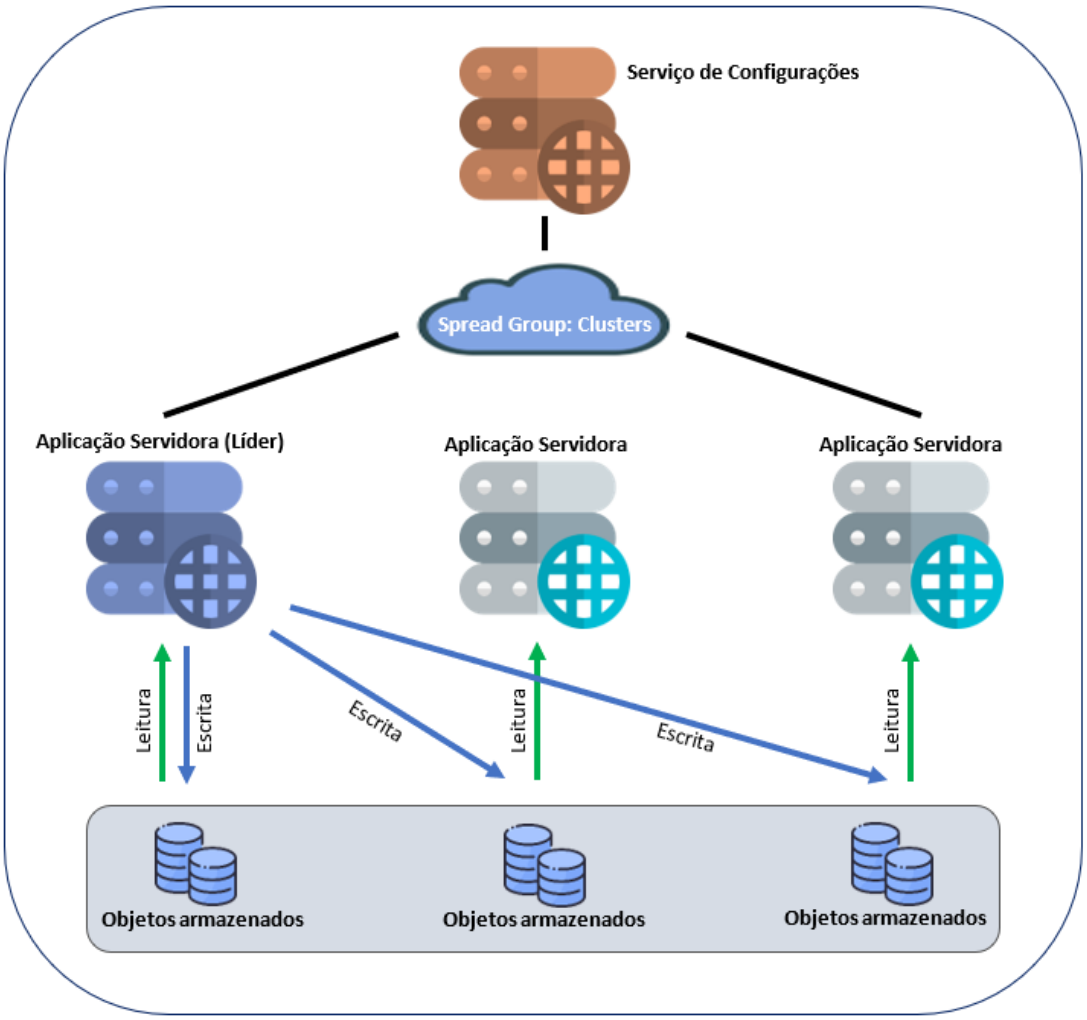


*Figura 1: Diagrama da arquitetura interna do sistema.*

## 2. Comunicação intra-cluster

Conforme referido na introdução deste relatório, um sistema distribuído pode ser, de uma forma simples, caracterizado como uma coleção independente de máquinas/instâncias que colaboram entre si e que da perspetiva dos utilizadores são vistas como um único sistema coerente.

Para que essa colaboração fosse possível, foi necessária a criação de um protocolo de sintaxe e semântica de mensagens, por forma a garantir a transmissão organizada de informação entre as aplicações servidoras através do Spread.

Assim sendo, serão de seguida apresentados os vários tipos de mensagens e, posteriormente, as situações em que as mesmas são usadas.

| DISCOVERY_REQ: | Private group name |
|---|---|

Mensagem enviada pelo líder a fim de descobrir quais são, naquele momento, as aplicações servidoras em operação no cluster. A mensagem contém no seu corpo o *private group name* do config_svc (*unicast address*) para que as eventuais aplicações lhe possam responder diretamente.

| DISCOVERY_ACK: | public ip | private ip | gRPC service port |
|---|---|---|---|

Resposta (unicast) a uma mensagem tipo DISCOVERY_REQ enviada ao serviço de configurações (config_svc). Contém dados de rede do membro do cluster.

| WRITE_REQ: | key | value |
|---|---|---|

Pedido de autorização enviado ao líder do grupo (unicast) para escrita local de um objeto.

| COMMIT: | key | value |
|---|---|---|

Mensagem de autorização, enviada do líder para o membro do cluster, (unicast) para escrita local de um objeto.

| READ_REQ: | key |
|---|---|

Caso não exista no servidor uma réplica local, é enviada esta mensagem para o grupo (multicast) a questionar pelo objeto com a *key* fornecida.

| READ_RPY: | key | value |
|---|---|---|

Resposta (unicast) a uma mensagem tipo READ_REQ. Se o membro do cluster que recebe o pedido também não tiver o objeto na sua réplica, o campo *value* é *null*.

| REMOVE_REQ: | key |
|---|---|

Pedido enviado ao líder do grupo (unicast) para remoção de todas as réplicas de objetos com a *key* fornecida.

| INVALIDATE_REQ: | key |
|---|---|

Ordem enviada pelo líder a todos os membros do cluster (multicast) para invalidação/remoção de réplicas de objetos com a *key* fornecida.

| SET_LEADER: | leader name |
|---|---|

Mensagem enviada pelo serviço de configurações (config_svc) de anúncio de novo líder do grupo. Contém o nome do líder sob a forma do seu privateGroup (endereço unicast Spread).

| UPDATE_REQ: | key |
|---|---|

Os membros do cluster enviam esta mensagem aquando do *startup* para atualizarem os valores associados às chaves que constam nas suas memórias secundárias.

| UPDATE_RPY: | key | value |
|---|---|---|

Resposta (unicast) a uma mensagem tipo UPDATE _REQ enviada por um membro do cluster que acaba de arrancar. Ao contrário do que acontece com mensagens do tipo READ_RPY, o valor nunca é null, pois as aplicações servidoras estão programadas apenas para responderem se tiverem o objeto armazenado na sua memória.

## 3. Comunicação cliente – serviço de configurações

Uma das partes mais importantes do sistema consiste nas interações entre aplicações clientes e serviço de configurações (config_svc) para que estes sejam atualizados quanto à lista de servidores disponíveis.

De seguida são explicados os detalhes sobre a forma como essas interações ocorrem.

### 3.1. Notificações de atualização do sistema

Os utilizadores acedem ao cluster para ler ou escrever dados por meio de uma aplicação Java 11. Inicialmente, os clientes necessitam saber quais as aplicações servidoras disponíveis para se poderem ligar a uma delas.
Essa informação é obtida pela invocação da operação *getClusterGroup* através de um stub não-bloqueante onde é passada a referência de um StreamObserver<type> para receção de notificações do serviço de configuração. Nessas notificações constam as listas atualizadas de aplicações servidoras disponíveis.

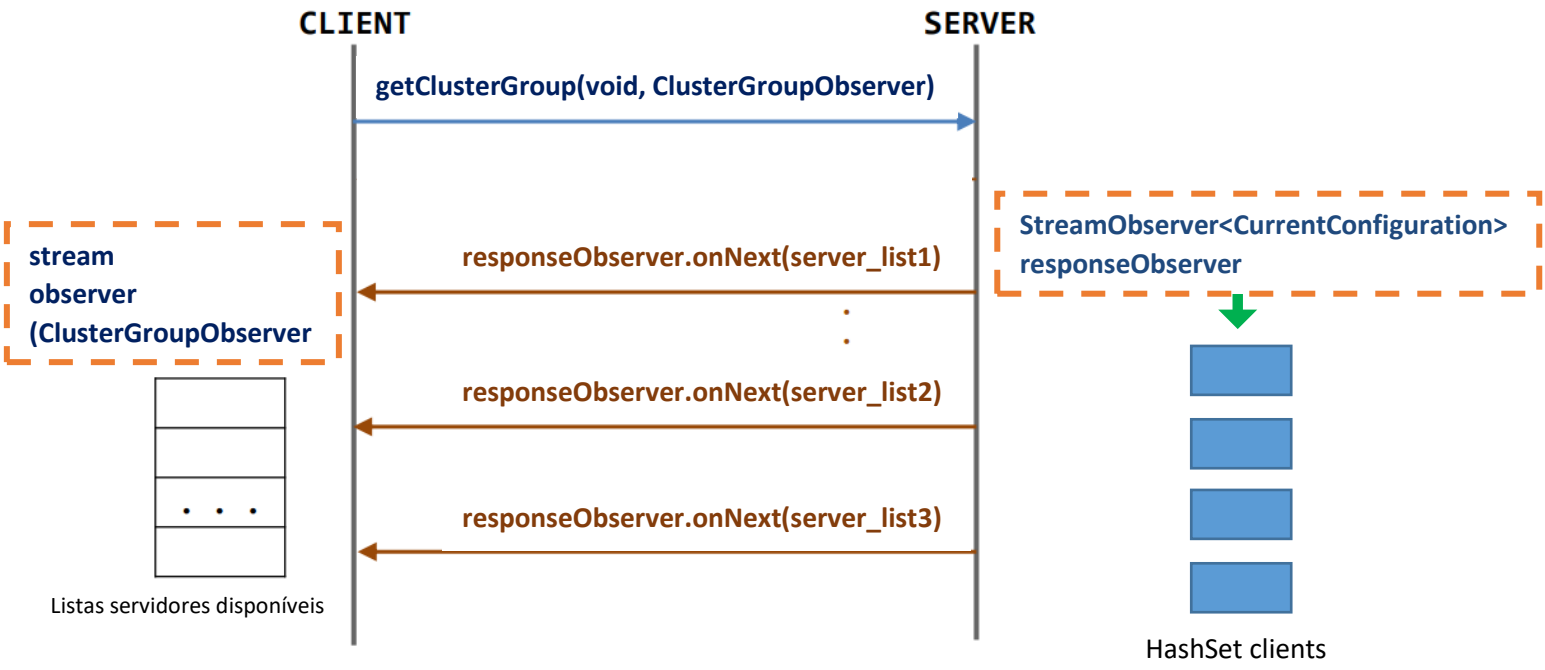


*Figura 2: Mecanismo de obtenção dinâmica de listas de servidores disponíveis.*

O serviço de configurações, por sua vez, armazena num HashSet, todos estes *stream observers* (SO).

Sempre que ocorra uma alteração na configuração do cluster (entrada ou saída de uma aplicação servidora), é feita uma iteração ao HashSet onde é invocado o método onNext() dos SO's de modo a enviar a nova configuração aos clientes – *server streaming*.

Enquanto um cliente estiver conectado está sempre pronto a receber notificações.
**Durante este processo de notificação, aproveita-se também para fazer o controlo de clientes conectados ou não-conectados.**
Numa situação em que uma aplicação cliente seja encerrada, as chamadas onNext() irão falhar resultando num tratamento de exceção por parte do servidor, onde o SO associado é retirado do HashSet.

O formato das mensagens que contêm a lista de servidores disponíveis é definido no contrato, escrito em *protocol buffers*.

```
message CurrentConfiguration {
     repeated string host_and_port = 1;
}
```

A adoção deste mecanismo de notificações levanta uma questão que se prende com o facto de, na verdade, nunca ser chamado o método onCompleted().

Uma solução mais simples para as notificações de atualização do sistema seria a de usar uma operação unária – Request/Response.

Dessa forma, os clientes inicialmente receberiam a lista de aplicações servidoras disponíveis e escolheriam uma para se conectarem. Assim, a necessidade de manter canais ativos durante todos o tempo de vida dos clientes seria minimizada e o serviço de configurações (config_svc) deixaria de ter de armazenar *stream observers*.

## 4. Comunicação cliente – aplicações servidoras

Os utilizadores podem efetuar a leitura ou escrita de objetos através da chamada dos métodos unários *Read* e *Write* do stub bloqueante de acordo com a definição de um serviço próprio para o efeito.

```
service MemoryService {
 rpc Read(Key) returns(KeyValuePair);
 rpc Write(KeyValuePair) returns(Void);

}
```

As mensagens *Key* e *KeyValuePair* apresentam o seguinte formato.

```
message Key {
 string key = 1;
}
```

```
message KeyValuePair {
 string key = 1;
 string value = 2;
}
```

```
message Void {

}
```

## 5. Listas aplicações servidoras disponíveis

O serviço de configurações mantém uma estrutura de dados dedicada e devidamente atualizada com as aplicações servidores que estão em funcionamento. Sempre que ocorrer um evento de "membership" no grupo *Spread* do cluster, quer seja entrada (JOIN) ou saída (LEAVE e DISCONNECT), este serviço automaticamente envia uma mensagem tipo DISCOVERY_REQ para o grupo, através do método `discoveryRequestHandler()`.

As aplicações servidoras estão programadas para, sempre que receberem uma mensagem deste tipo, responderem com um DISCOVERY_ACK contendo os seus dados de rede.

O serviço de configurações ao receber todas estas respostas/ackonowledges, procede ao seu processamento no método `discoveryAckHandler()`:

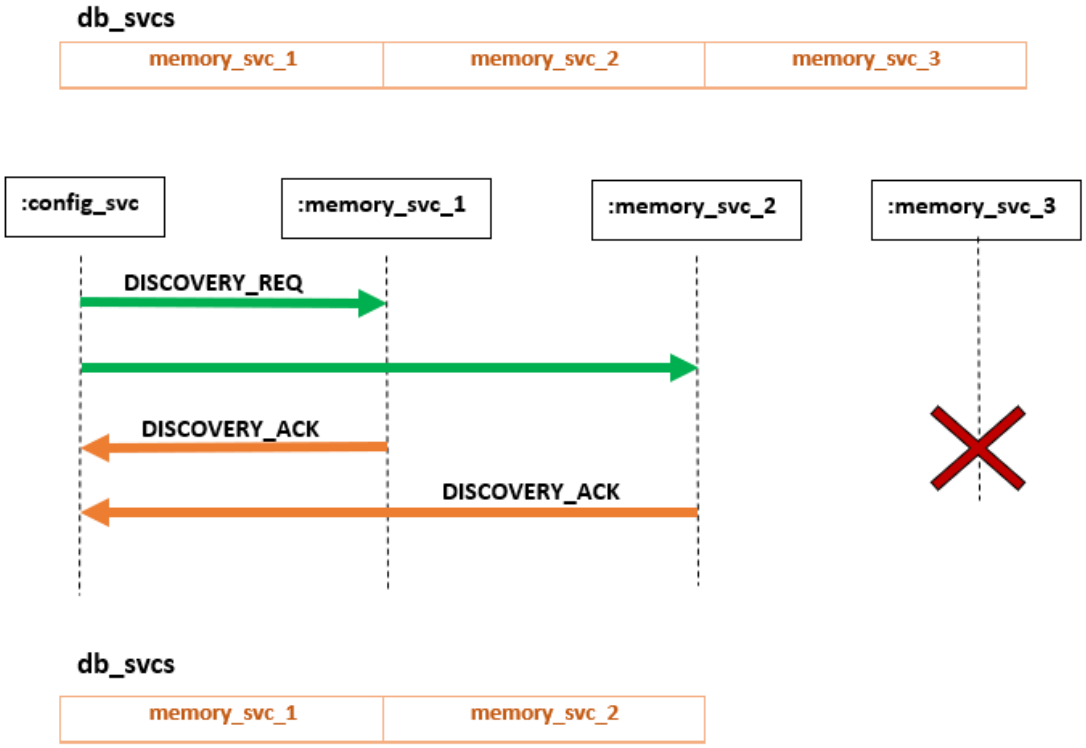


*Figura 3: Mecanismo de atualização de listas de servidores disponíveis.*

Se a mensagem de ACK for de membros já conhecidos pelo config_svc, a mesma é ignorada. Caso contrário, procede-se à adição do *hostname* do novo servidor à estrutura de dados.

A remoção de membros do cluster da estrutura de dados é feita imediatamente aquando da receção de uma mensagem Spread de membership tipo LEAVE ou DISCONNECT.

Este **processo de "discovery" marca** também o **início da eleição de um novo líder do grupo**, conforme se verá adiante.

**Quando o serviço de configurações** (config_svc) **está a arrancar pela primeira vez**, **o método `discoveryRequestHandler()` também é invocado** para que, numa situação em que o config_svc arranque com aplicações servidoras já em funcionamento, possa coletar a devida informação do cluster no presente.

## 6. Mecanismo de Consenso e Replicação

O mecanismo de consenso e replicação implementado foi inspirado no famoso algoritmo de consenso *Raft*. Segundo este algoritmo, se se considerar um grupo de *Database servers*, um destes irá assumir o papel de "líder" na medida em que todas as alterações do sistema passarão por ele.

De igual forma, implementou-se um sistema idêntico neste projeto em que se tira proveito da existência do serviço de configurações e do facto deste se encontrar no mesmo grupo S*pread* que as aplicações servidoras.

Conforme explicado no ponto anterior, sempre que ocorre um evento de *Membership*, nomeadamente uma entrada ou saída de uma aplicação servidora, o *config_svc* atualiza a sua "lista" de membros do cluster em funcionamento para posteriormente notificar os clientes.
Esta atualização é feita com base no envio e receção de mensagens tipo DISCOVERY_REQ e DISCOVERY_ACK.

Assim sendo, optou-se por aproveitar este processo para despoletar a eleição de um novo líder do grupo – sempre que ocorre um evento de *membership*, realiza-se uma nova eleição.

O config_svc elege como líder o membro do cluster cujo DISCOVERY_ACK foi rececionado primeiro.

De seguida, o resultado da eleição é anunciado pelo config_svc ao cluster inteiro através do envio de uma mensagem tipo SET_LEADER que contém o *hostname* do líder, coincidente com o seu privateGroup (unicastAddress).

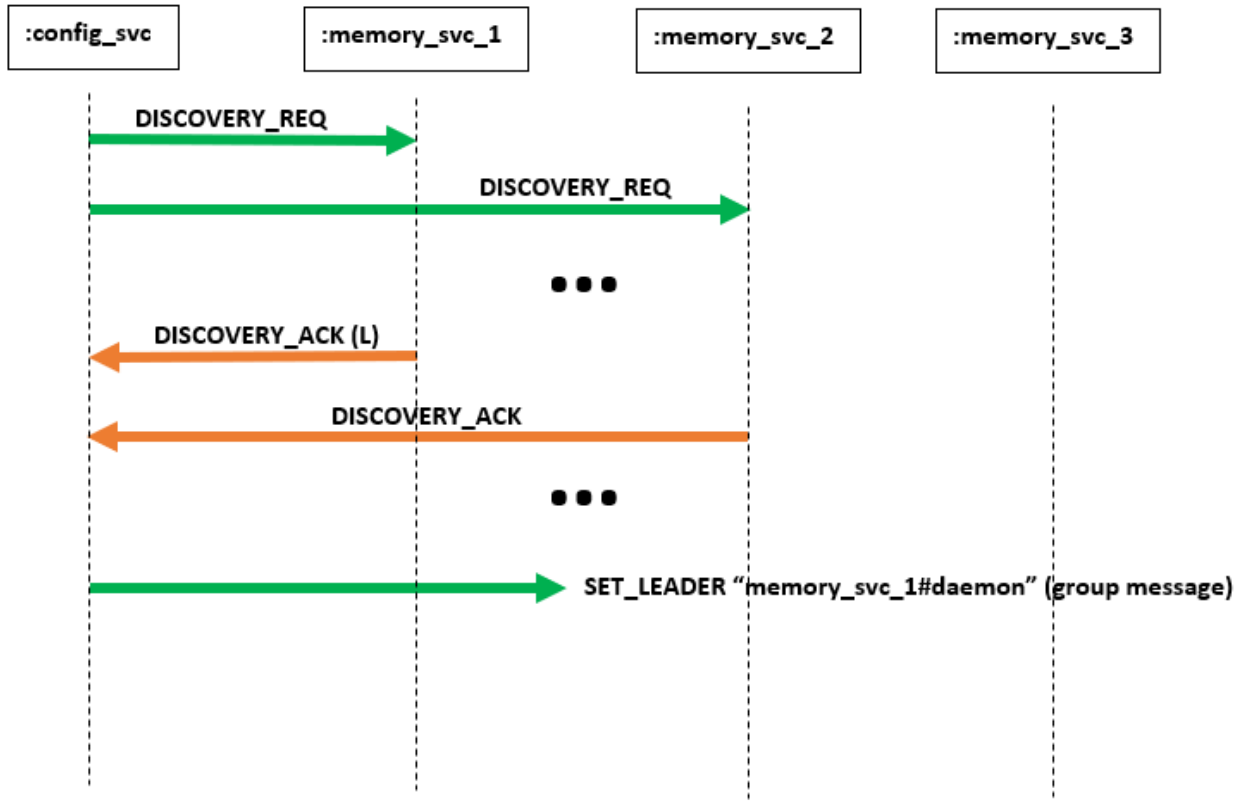


*Figura 4: Mecanismo de eleição de líder.*

Nas aplicações servidoras, uma variável *boolean*, que funciona como uma espécie de *flag*, inicialmente muda de estado no sentido de indicar a ausência de líder no sistema, pois está prestes a iniciar-se um processo de eleição.

Quando os membros do cluster recebem as mensagens SET_LEADER, guardam o hostname/unicastAddress do novo líder e mudam o estado da *flag* mencionada anteriormente para *existência de líder no sistema*.

Conforme se pode ver, no fundo, o serviço de configurações acaba por funcionar como uma espécie de "árbitro" que gere o processo de eleição.

Apesar da simplicidade desta solução, existem algumas desvantagens com a sua adoção, nomeadamente:

- É implícito que o sistema só pode funcionar na sua plenitude se o config_svc estiver em operação – **as operações de escrita deixam de ser possíveis**;
- Nos momentos em que o sistema está todo ele a efetuar o *startup*, ocorrem várias eleições sucessivas desnecessárias.

## 6.1. Processamento de alterações ao sistema.

Um dos requisitos funcionais do projeto é o de que cabe ao servidor que recebe o pedido do cliente, desencadear as ações com vista a garantir a consistência.

### 6.1.1. Leitura de objetos

Suponha-se o cenário simples em que um servidor recebe um pedido de leitura do valor de um objeto com chave *abcxyz*.

Se existir no servidor uma réplica local, o valor do objeto é devolvido imediatamente.

Caso não exista, é enviada uma mensagem *Spread* tipo READ_REQ, contendo a chave no seu corpo, para todo grupo *Cluster*.

As aplicações servidoras estão programadas para que sempre que recebam uma mensagem deste tipo efetuem a devida pesquisa na sua memória e respondam com uma mensagem tipo READ_RPY.

**Se o membro do cluster que recebe o pedido também não tiver o objeto na sua réplica, o campo *value* é *null*.**

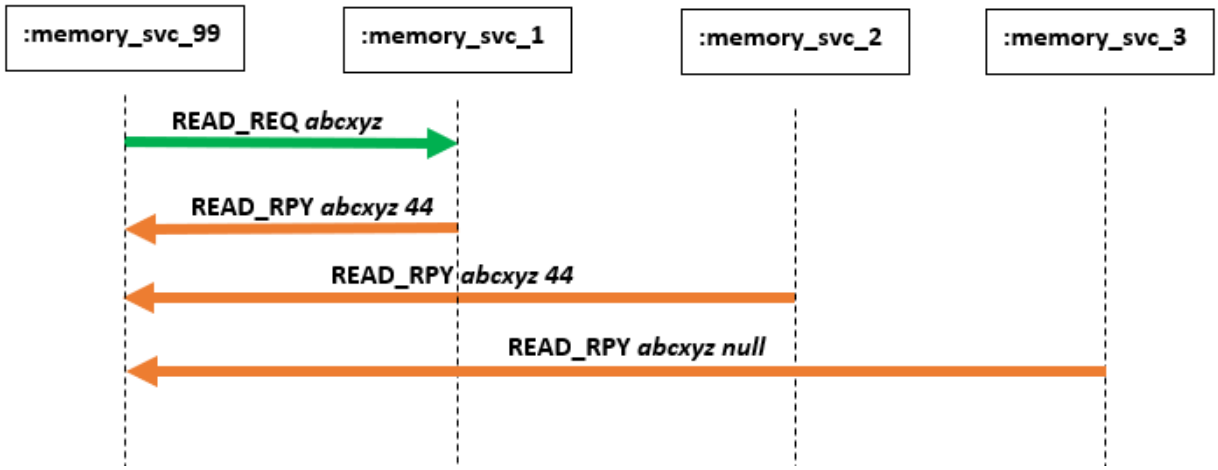


*Figura 5: Mecanismo de leitura de valor de um objeto para a situação em que o mesmo não consta no servidor que recebeu o pedido.*

Com isto, levantaram-se duas questões durante o desenvolvimento desta funcionalidade:

- Como "marcar" os pedidos, ou seja, como saber a quais READ_REQ correspondem os READ_RPY's recebidos ?
- Como saber se um pedido já foi "atendido" ou não?

A solução adotada para a resolução deste problema foi a de implementar um mecanismo de *logs* onde são guardados os pedidos que foram respondidos.

O emissor fica à espera de que a resposta seja registada nesse log no método `ListenForReply()`.

Quando a resposta chega é adicionada a entrada ao log e o *callback* registado é chamado.

### 6.1.2. Escrita de objetos

Quando um servidor recebe um pedido de escrita, ocorre todo um conjunto de ações.
Iniciamente, é feita uma pesquisa no sentido de averiguar se existe alguma cópia do objeto no cluster.

Se não existir, é feita a escrita localmente. Para que tal aconteça, é feito um pedido WRITE_REQ ao líder do grupo que terá como resposta um COMMIT.

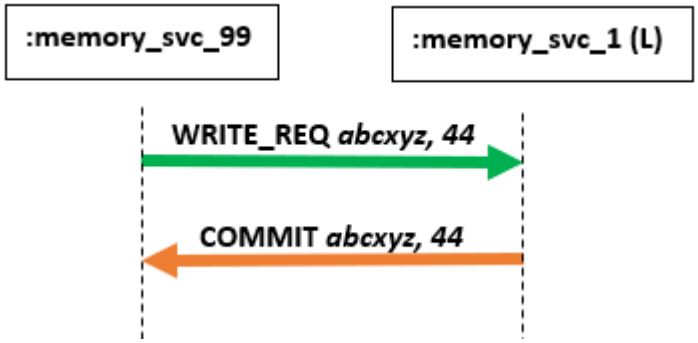


*Figura 6: Escrita de objeto em situações onde o mesmo não existe no cluster.*

Nesta situação, o líder do grupo escreve o objeto imediatamente.
Caso contrário, é desencadeado um processo de invalidação dessa(s) réplica(s):

1. A aplicação servidora envia uma mensagem ao líder REMOVE_REQ a pedir a invalidação de todas as réplicas com a chave em questão;

2. O líder do grupo, ao receber essa mensagem, envia uma ordem para todo o cluster, o líder idem, sob a forma de uma mensagem do tipo INVALIDATE_REQ, para que sejam invalidadas todas as réplicas com a chave em questão.

   As aplicações servidoras estão programadas para o fazerem sempre que receberem uma mensagem deste tipo.

3. Escrita do objeto localmente.

Nesta situação, se for o líder do grupo quem tenha recebido o pedido de escrita, o processo de invalidação, naturalmente, inicia-se no passo 2.

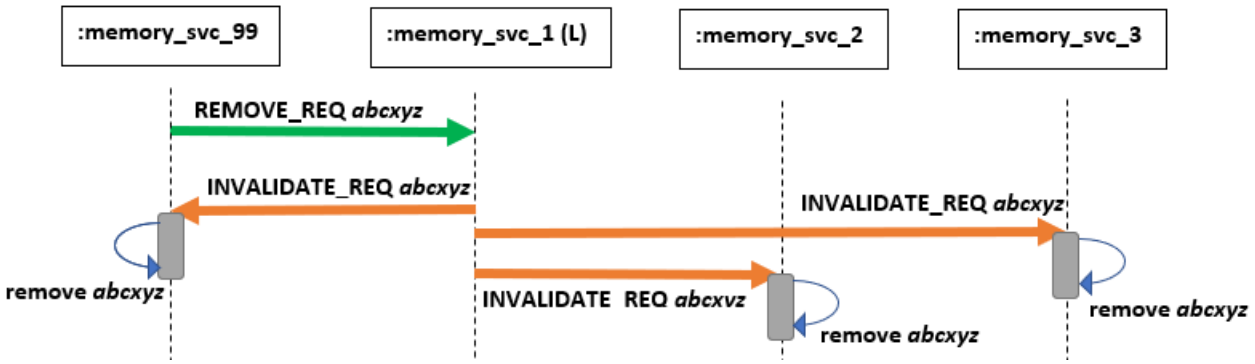


*Figura 7: Mecanismo de invalidação de réplicas.*

Deste modo, assegura-se o consenso na invalidação das réplicas e um novo objeto é escrito sem conflitos.

No entanto, **pode surgir um problema com esta solução** que reside no facto dos servidores não enviarem um sinal de *feedback* ao líder a informar que a remoção dos objetos ocorreu com sucesso.

Na eventualidade de ocorrer um problema na remoção, resultando na manutenção do objeto, dois clientes poderão obter valores diferentes para a mesma chave.

### 6.1.3. Reentradas de servidores

Seja a seguinte figura a representação de um cenário em que estão três servidores em funcionamento com vários pares (chave, valor) guardados nas suas réplicas locais.

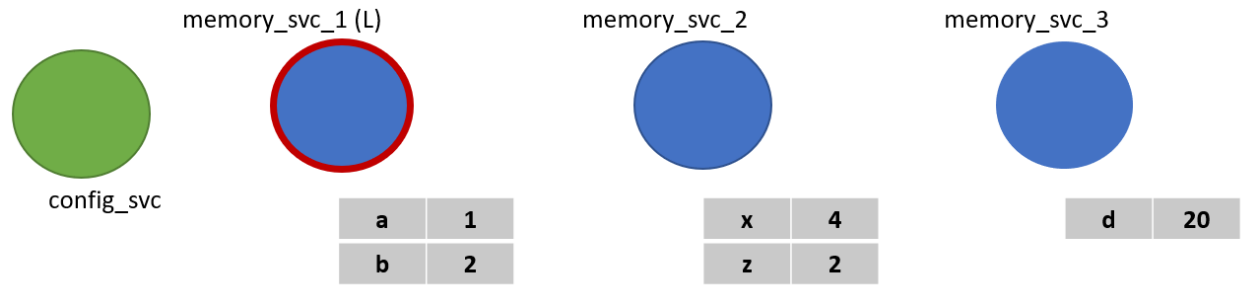


*Figura 8: Exemplo de cenário de normal funcionamento do cluster.*

Considere-se que, por motivo técnico, por exemplo, o servidor *memory_svc_db_3* se encontra em falha e fica desconectado do grupo.

Depois de ocorrer um novo processo de eleição, o servidor *memory_svc_1* mantém-se como líder.

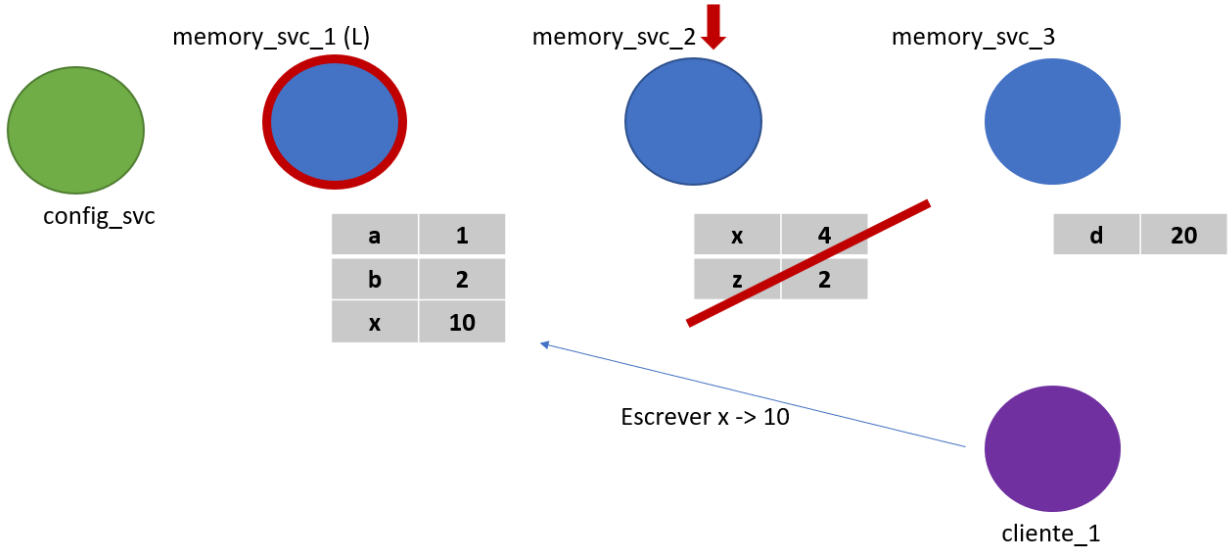


*Figura 9: Ocorrência de falha de um dos servidores.*

Se um cliente pretender escrever um par que tenha chave "x", noutro servidor, não haverá invalidação de réplicas, pois mais nenhuma outra instância contém um par com essa chave. Isto constitui um problema na medida em que quando o *memory_svc_db2* volte a juntar-se ao grupo, irão existir dois pares com a mesma chave, mas com valores diferentes.

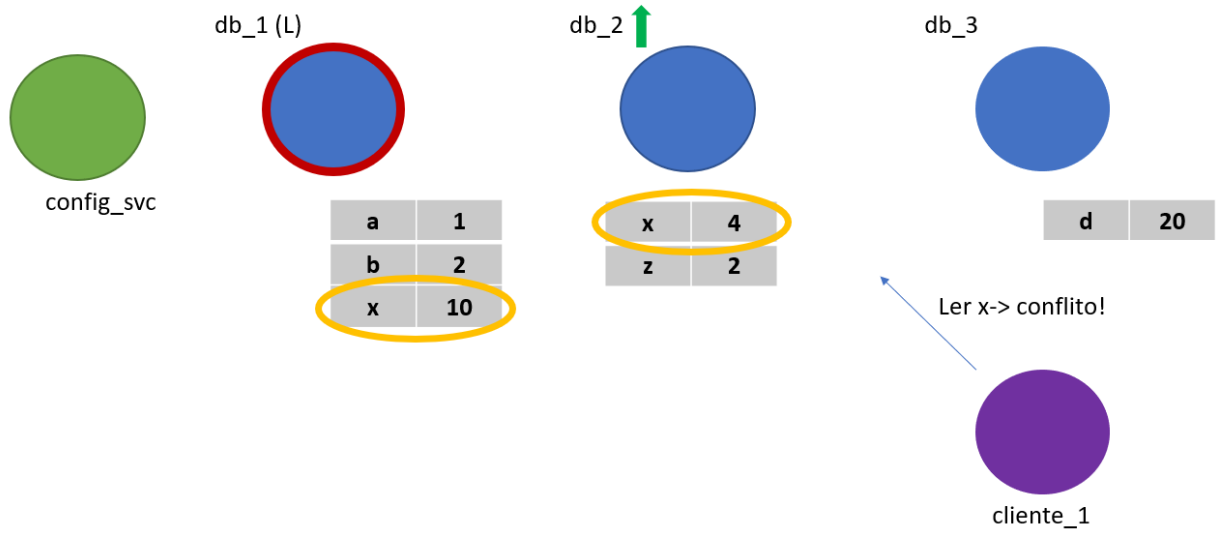


*Figura 10: Situação de conflito no cluster.*

A solução que se adotou para este problema foi a de obrigar os servidores, durante o processo *startup*, a:

1. Efetuarem o *setup* no sentido de **se juntarem ao grupo *Spread* do cluster**;
2. Iterarem sobre todas as suas chaves e **enviarem**, para cada uma, um pedido tipo **UPDATE_REQ**.
   As aplicações servidoras estão programadas para procurarem e devolverem os valores associados sempre que receberem uma mensagem deste tipo. Se não tiverem nenhum objeto, não é enviada qualquer resposta;
3. O servidor que recebe as respostas aos UPDATE_REQ's, **efetua a escrita dos eventuais novos valores na sua memória**, caso contrário trata-se apenas de um *rewrite* sem efeito;
4. **Terminado este passo**, após um intervalo de tempo estipulado, é então feito o **setup da componente gRPC** para que o servidor já possa receber eventuais pedidos de clientes.

# CONCLUSÃO

Um sistema distribuído pode ser, de uma forma simples, caracterizado como uma coleção independente de máquinas/instâncias que colaboram entre si e que da perspetiva dos utilizadores são vistas como um único sistema coerente.

Uma vez que no âmbito deste projeto surgiu a necessidade de assegurar essa colaboração, criou-se de um protocolo de sintaxe e semântica de mensagens, por forma a garantir a transmissão organizada de informação entre as aplicações servidoras através do Spread.

Os clientes obtêm, de forma dinâmica, recebem listas contendo os servidores disponíveis por meio do uso de server *streaming* via gRPC.

A leitura ou escrita de objetos é feita através da chamada dos métodos unários gRPC *Read* e *Write* do stub bloqueante de acordo com a definição de serviços próprios para o efeito.

Foi implementado um mecanismo de consenso e replicação inspirado no famoso algoritmo de consenso *Raft* que tira proveito da existência do serviço de configurações e do facto deste se encontrar no mesmo grupo S*pread* que as aplicações servidoras.
Sempre que ocorre entrada ou saída de uma aplicação servidora, o *config_svc* atualiza a sua "lista" de membros do cluster em funcionamento para posteriormente notificar os clientes.
Esta atualização é feita com base no envio e receção de mensagens tipo DISCOVERY_REQ e DISCOVERY_ACK.

O config_svc elege como líder o membro do cluster cujo DISCOVERY_ACK foi rececionado primeiro.

A importância do papel de "líder" reside no facto de ser por essa instância que passam todas as alterações do sistema – escrita e invalidação de réplicas. A leitura de local de objetos não envolve o líder.

Para situações em que ocorra saída e entrada de servidores no grupo, foi implementado um mecanismo de "sincronização" por forma a não existirem conflitos de dados.

A realização deste trabalho permitiu de certa forma ganhar a noção de que em sistemas distribuídos existem vários cenários de falha/erro que necessitam ser considerados. A gestão de conflitos de dados é também importante para que o sistema aos "olhos" do utilizador/cliente continue a ser visto como uma só entidade coerente.

Os compassos de espera entre operações (como, por exemplo, envio e receção de mensagens),que noutros domínios podem ser desprezados, merecem especial atenção em computação distribuída, pois é o suficiente para haver diferenças de valores observados por dois clientes, por exemplo.