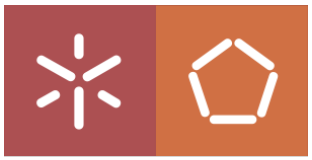
Universidade do Minho
Escola de Engenharia

Paradigmas de Sistemas Distribuídos + Sistemas Distribuídos em Grande Escala
Junho 2023



Universidade do Minho
Escola de Engenharia

Armazenamento em Grande Escala

Alexandre Martins (PG50168), Bruno Pereira (PG50271), Rui Braga (PG50743)

Armazenamento em Grande Escala

Alexandre Martins (PG50168), Bruno Pereira (PG50271), Rui Braga (PG50743)

Junho 2023

1 Introdução

Atualmente, uma organização depende do sistema de armazenamento dos seus dados, e a necessidade de criação de sistemas de armazenamento eficientes e escaláveis tornou-se ainda mais notória. Um sistema de armazenamento em grande escala representa uma mudança de paradigma face aos sistemas de armazenamento tradicionais que dependem de uma arquitetura centralizada.
Neste relatório vai ser explicada a implementação de um sistema de armazenamento em grande escala que tem como principais objetivos apresentar coerência causal e alta disponibilidade para os utilizadores.

2 Implementação

2.1 Servidores de Sessão

Os servidores de sessão são o ponto de ligação entre os clientes e o sistema, sendo responsáveis por atender os pedidos dos clientes, e consultar os servidores de dados para concretizar os pedidos. Os servidores de sessão mantêm, de forma não persistente, o contexto de cada cliente, de modo a que os pedidos possam ser realizados mantendo coerência causal. Para evitar que certos clientes monopolizem os recursos dos servidores, degradando assim o desempenho do sistema, foi implementado um mecanismo que permite limitar os clientes que ”abusem” na quantidade de pedidos feitos.

2.1.1 Arquitetura dos servidores de sessão

O servidor de sessão foi desenvolvido em *Erlang*, de modo a tirar partido do baixo custo na criação de novos processos e no baixo custo do *context-switch* entre estes. Tendo em conta esta caraterística, foi possível desenvolver código com caráter sequencial, para as diversas componentes necessárias para o funcionamento do servidor de sessão. Além disso, permite-nos a criação de um processo *Erlang* por cliente, o que facilita substancialmente a lógica a ser implementada, já que o processo atribuído a um cliente, pode gerir a informação sobre o cliente de forma encapsulada, e sem que outros processos possam interferir.

O servidor de sessão possui 6 tipos de atores:

- **Acceptor** - ator responsável por ficar à escuta de conexões TCP, aceitando-as, e ”criando” um processo para atender o cliente (*Client Responder*). Na verdade, um *Acceptor* cria um novo processo *Acceptor*, e deixa o atual converter-se num *Client Responder*.

- **Client Responder** - ator que implementa a lógica necessária para fazer a autenticação do cliente, e atender os seus pedidos de leitura e escrita. Pede ao ator **Users State** para autenticar o cliente se este ainda não se encontrar autenticado (em nenhum servidor de sessão). Caso o cliente já esteja autenticado noutro servidor de sessão, recusa o pedido de autenticação, e fecha o *Socket*. Este ator, também é responsável por consultar o ator *Shared Cache*, para verificar existência de versões em *cache* antes de encaminhar o pedido para o ator *Data Gateway*, diminuindo assim a carga nos servidores de dados. Quando recebe novas versões do ator *Data Gateway*, envia-as para a *Shared Cache*.

- **Shared Cache** - ator responsável por gerir uma *cache LRU (Least Recently Used)*. Atende pedidos de leitura/escrita da/para a *cache*.

- **Limiter** - ator responsável por manter registo dos clientes limitados (*throttled*) localmente, e libertar um destes periodicamente. Este ator quando recebe um pedido de um *Client Responder* para limitar um cliente, envia também esta informação para o ator *User State* para que este possa partilhar a informação com os restantes servidores de sessão. Ao partilhar esta informação, um cliente que tenha acabado de ser limitado, não pode escapar a esta situação ao conectar-se a outro servidor de sessão.

- **Users State** - ator que guarda informação sobre os clientes que estão autenticados e limitados. Este estado é partilhado pelos diversos servidores de sessão. Para manter este estado coerente entre os diversos servidores de sessão, é necessário o envio periódico do estado para um dos *brokers* que conectam os servidores de sessão. O *broker* que receber o estado, é então responsável de fazer ”broadcast” do estado para todos os servidores de sessão.

- **Data Gateway** - ator que encaminha os pedidos para os servidores de dados. Esta comunicação é explicada mais à frente no relatório.

Na figura 1, presente nos anexos, é possível verificar um diagrama que resume as interações entre os atores.

2.2 Replicação do estado dos servidores de sessão

2.2.1 CRDTs

Para o ator *User state* manter o estado partilhado atualizado e coerente entre os vários servidores de sessão foram utilizadas *state-based CRDTs*. O estado a ser partilhado pode ser representado por dois *sets*, dito isto, a partilha do estado de uma forma que respeite a coerência causal e que seja convergente, o grupo optou por fazer a implementação de uma Causal CRDT (com *Dot Store* e *Causal Context*). A estrutura de dados implementada foi o *Observed-Removed Set(ORSet)*. Depois de definido o estado partilhado como um par de *ORSets*, falta a partilha do estado entre os servidores de sessão. Cada servidor de sessão, periodicamente, partilha o seu estado local com os outros servidores de sessão (é enviado todo o estado). Quando um servidor recebe um estado, este realiza a operação de *merge* entre os dois estados, de forma a obter a versão mais atualizada entre os dois estados.

2.2.2 Topologia

A topologia utilizada para a distribuição do estado pode ser observada na figura 2 que está presente nos anexos. Utilizando esta topologia é possível os servidores de sessão entregarem a um *broker* (dos muitos que podem existir nesta camada) enviando de forma assíncrona, o estado para o *broker*, através do *socket* do tipo *DEALER*, para o socket do tipo *ROUTER* presente no *broker*. Todos os

servidores de sessão, para além do *socket DEALER*, possuem um socket *SUB*, que utilizam para subscrever a receção do estado. Ao utilizar o padrão *PUB-SUB* é possível os *brokers* fazerem *"broadcast"* dos estados, garantindo que este chega a todos os servidores de sessão. O facto de utilizar esta topologia, permite também o aumento de *brokers*, de forma a permitir balanceamento da carga, e também evitar um ponto único de falha.

2.3 Limitador de carga

Como referido anteriormente, para impedir que clientes perturbem o desempenho do sistema, os servidores de sessão implementam um limitador de carga. Para implementar este limitador de carga, o processo *Erlang* responsável pela comunicação com o cliente (*Client Responder*) monitoriza a média de pedidos por segundo nos últimos 60 segundos. Esta monitorização é realizada utilizando uma estrutura de dados que o grupo designou por *circular buffer*. O *buffer* é atualizado no fim de cada pedido. De seguida, é feita a computação da média dos últimos 60 segundos, e se o valor tiver superado o limite *LIMIT*, então o cliente é *throttled* (limitado), i.e., o limite atual passa a ser *BASE*, que deve ser um valor consideravelmente inferior a *LIMIT*. Para limitar o cliente, é utilizada a opção *active, once* do *socket TCP*, que permite o controlo de fluxo. Esta opção apenas é reativada, utilizando *inet:setopts()*, quando a média de pedidos por segundo nos últimos 60 segundos voltar a ser inferior ao limite atual. A informação de que o cliente foi limitado é partilhada com os outros servidores de sessão, para que este continue limitado, caso decida conectar-se a outro servidor.

2.3.1 Circular Buffer

O *circular buffer* consiste, essencialmente, num *buffer* com um tamanho fixo de *slots*. Cada *slot* cobre um período de tempo que corresponde à divisão dos 60 segundos pelo número de *slots*. Os *slots* são iniciados com valor 0, e cada vez que um pedido acontece no intervalo correspondente a esse slot, o valor desse *slot* é incrementado. O primeiro passo para atualizar o *buffer* é calcular o índice (não circular) ao qual o *timestamp* do momento atual corresponde. Com este índice é possível calcular o índice circular, i.e., o índice do slot que deve ser atualizado. O índice não circular do pedido atual, aliado ao índice não circular da atualização anterior, permite calcular os slots que devem ser reiniciados (valor passar para 0). A **média aproximada** dos pedidos realizados nos últimos 60 segundos, pode então ser calculada somando os valores de todos os *slots* e posteriormente, dividindo o resultado dessa soma pelo número total de *slots*. Nas figuras 3 e 3 em anexo é possível ver exemplo de algumas iterações do algoritmo que atualização.

2.4 Diminuição dos pedidos aos servidores de dados

Os servidores de sessão possuem uma *cache LRU*. Esta *cache*, com tamanho limitado configurável, guarda versões de chaves, para que se possam evitar consultas desnecessárias aos servidores de dados. Esta *cache* é consultada quando existem dependências, ligadas a uma chave que se requisitou numa transação de leitura, que têm versão superior à versão de uma mesma chave requisitada na transação. A implementação desta *cache* utiliza duas estruturas de dados. A primeira é *gb_trees()*, que corresponde a uma árvore ordenada, e que permite a organização das versões em função do *timestamp* em que foram inseridos/acedidos. A segunda é um *map()* para tornar os *lookups* mais eficientes.

2.4.1 Interface de Dados - Algoritmo de COPS

Os servidores de sessão vão disponibilizar aos clientes dois tipos de pedidos: transações de leitura e escritas independentes. Para realizar estes pedidos vão precisar de uma interface para comunicar

com os servidores de dados, de modo a conseguirem atender os pedidos de escrita e leitura mantendo sempre a consistência causal entre as várias operações.
Para isso, criou-se o módulo de interface de dados que implementa o algoritmo de COPS de maneira a que se possam realizar transações de leitura obtendo dados consistentes (ou seja, sem a falta de depências de escrita). Cada cliente vai ter um contexto de COPS onde serão armazenadas todas as dependencias relevantes aos próximos pedidos desse cliente. Sempre que um cliente realiza uma leitura, as chaves e a versão lida são armazenadas no contexto de COPS sobre a forma de dependencias (as dependências neste contexto são pares (chave, valor)). Por outro lado, quando um cliente faz uma escrita, esta escrita vai ter como dependencias todas aquelas que estiverem armazenadas no contexto COPS, que serão de seguida substituídas por uma única dependencia que será a chave e a versão da escrita realizada.
Deste modo, ao realizar uma transação de leitura, a interface de dados fica responsável por verificar se todas as dependencias existentes nas chaves lidas estão presentes, se sim, devolve os resultados ao cliente, caso contrário vai pedir aos servidores de dados as depêndencias que faltam.

2.5 Servidores de Dados

Um servidor de dados é identificado através de um código hash gerado aleatoriamente, que se encontra entre *Integer.MIN_VALUE* e *Integer.MAX_VALUE* e serve para identificar o servidor no *broker* de dados com quem um servidor comunica através de um socket ZeroMQ do tipo DEALER.

Além disso, o armazenamento de dados nestes servidores é feito usando um algoritmo de *consistent hashing*, que consiste em gerar um código hash para cada chave de acordo com o método *hashCode*, nativo de JAVA para o tipo *String*, cujo resultado estará compreendido entre os valores, *Integer.MIN_VALUE* e *Integer.MAX_VALUE*, sendo que o valor e metadados associados à chave serão, guardados no nodo de dados que se situa imediatamente antes ao seu código.

Os servidores de dados armazenam todos as chaves cujos códigos hash se situam entre o seu código e o código do servidor seguinte. Cada chave tem associada a si um conjunto de versões que, por sua vez, mapeiam os valores e as dependências (par chave-versão). Ou seja, para uma certa chave 'asd' que gere um código hash (de exemplo) 123, caso seja feita uma primeira escrita sem dependências do valor 42 e depois for feita uma escrita do valor 43 a depender da 3ª versão de outra chave 'dsa', o servidor de dados fica:

Código da Chave	Versão	Valor	Dependências
123	1	42	[]
	2	43	[('dsa', 3)]

É possível adicionar dinamicamente novos servidores de dados ao sistema e integrá-los através de um protocolo muito simples, que consiste em fazer um pedido ao servidor que antecede o servidor a inserir e selecionar as chaves que devem ser movidas, bem como os seus respetivos metadados. Dito isto, não é feita replicação dos dados e não são acauteladas falhas no funcionamento dos servidores, sendo que o único *backup* de dados que acontece é no protocolo de integração de um novo nodo e acontece a cópia de informação para esse novo nodo, pois, essa mesma informação, não é eliminada do nodo antigo.

2.5.1 Comunicação entre Servidores de Sessão e de Dados

A comunicação entre servidores de sessão e servidores de dados é feita através de um *broker* existente nos servidores de dados com um socket ZeroMQ do tipo ROUTER, que recebe pedidos vindos dos servidores de sessão e os dirige para os servidores de dados respetivos e também, faz a gestão de possíveis mensagens internas aos servidores de dados.

Sendo assim, as mensagens que um servidor de sessão envia têm 5 possíveis elementos: 'INNER' ou 'OUTER' que indica se a mensagem é interna aos servidores de dados ou não (inner), ID do nodo de onde vem a mensagem (nodeID), ID do cliente que fez o pedido (clientID), tipo (type) e contéudo da mensagem (content). Estes elementos ficam separados por '!'.

Formato de uma mensagem enviada ao broker dos servidores de sessão:

```
{inner}!{nodeID}!{clientID}!{type}!{content}
```

A comunicação feita entre os servidores de sessão e os servidores de dados é, então, feita através destes 3 pedidos:

- Pedido de Escrita

  O contéudo de uma mensagem deste tipo consiste num tuplo (chave, valor, dependências) que é representado, ao separar cada um dos 3 elementos por ';' e, realça-se também que, as dependências são pares (chave, versão), representadas no seguinte formato: [k1,v1/k2,v2/.../kn,vn]. De resto, a mensagem é 'OUTER' pois vem de um ambiento externo aos servidores de dados e o nodeID é negligenciável, pois coincide com a identidade do socket ('ss1', no exemplo).

  Como resposta é depois enviada uma mensagem do tipo 'write_ans' com a versão que corresponde à escrita que foi efetuada.

  Exemplo (SD - servidor de dados):

  ```
  pedido:       OUTER!!cliente1!write!abc;123;[{asd,1}/{dsa,3}]
  broker -> SD: INNER!ss1!cliente1!write!abc;123;[{asd,1}/{dsa,3}]
  SD -> broker: INNER!ss1!cliente1!write_ans!3
  resposta:     OUTER!!cliente1!write_ans!3
  ```

- Pedido de Leitura

  Este pedido é feito com o intuito de ler a versão mais recente de determinadas chaves e o seu contéudo de uma mensagem deste tipo é sequência das chaves a ser lidas separadas por ';'. A mensagem é novamente 'OUTER' e o id do nodo é igualmente negligenciável.

  Como resposta são depois enviadas várias mensagens do tipo 'read_ans', uma por cada chave, contendo chave, valor mais recente, versão mais recente e respetivas dependências, por ordem (chave, valor, versão, dependências).

  Exemplo (SD - servidor de dados):

  ```
  pedido:       OUTER!!cliente2!read!key1;key2;key3
  broker -> SD: INNER!ss2!cliente2!read!key1
  SD -> broker: INNER!ss2!cliente2!read_ans!key1;value1;1;[]
  resposta:     OUTER!!cliente2!read_ans!key1;value1;1;[]
  ```

- Pedido de Leitura de uma versão específica

  O contéudo de uma mensagem relativa a este pedido consiste num tuplo (chave, versão), cujos elementos vão separados por ';'. Como os outros 2 pedidos, a mensagem é 'OUTER' e o nodeID, negligenciável.

  Como resposta é envia-e uma mensagem do tipo 'read_version_ans' que contém um tuplo (chave, valor, versão), cujos elementos são representados, também, separados por ';'.

  Exemplo (SD - servidor de dados):

```
pedido:       OUTER!!cliente3!read_version!asd;1000
broker -> SD: INNER!ss3!cliente3!read_version!asd;1000
SD -> broker: INNER!ss3!cliente3!read_version_ans!asd;42;1000
resposta:     OUTER!!cliente3!read_version_ans!asd;42;1000
```

2.6 Aplicações Cliente

Foram feitas duas aplicações cliente para testar o sistema de diferentes formas e ambas são sequenciais, na medida em que, depois de fazer um pedido, só podem fazer outro, depois de já terem obtido resposta ao primeiro pedido.

Uma aplicação é interativa e tem o intuito de demonstrar o funcionamento do programa de acordo com a vontade do utilizador e outra permite emitir vários pedidos, que são gerados aleatoriamente e são espaçados por um certo intervalo de tempo ditado pelo utilizador, com o objetivo de verificar como se comporta o sistema com diferentes níveis de carga.

3 Conclusão

A implementação de um sistema de armazenamento a grande escala oferece várias vantagens em vertentes como, eficiência no acesso a dados e escalabilidade do seu armazenamento. A primeira é conseguida, principalmente, através do uso de técnicas como cache nos servidores de sessão e a existência de vários servidores de sessão para diferentes clientes e a segunda é conseguida através do particionamento dos servidores de dados, que permite acomodar os dados existentes enquanto que os volume de dados continuam a crescer.

Contudo, numa aplicação real de um sistema deste tipo, seria crucial ter, também em conta a manutenção e robustez do sistema, aspetos que não foram tidos como prioridade neste projeto.

4 Anexos

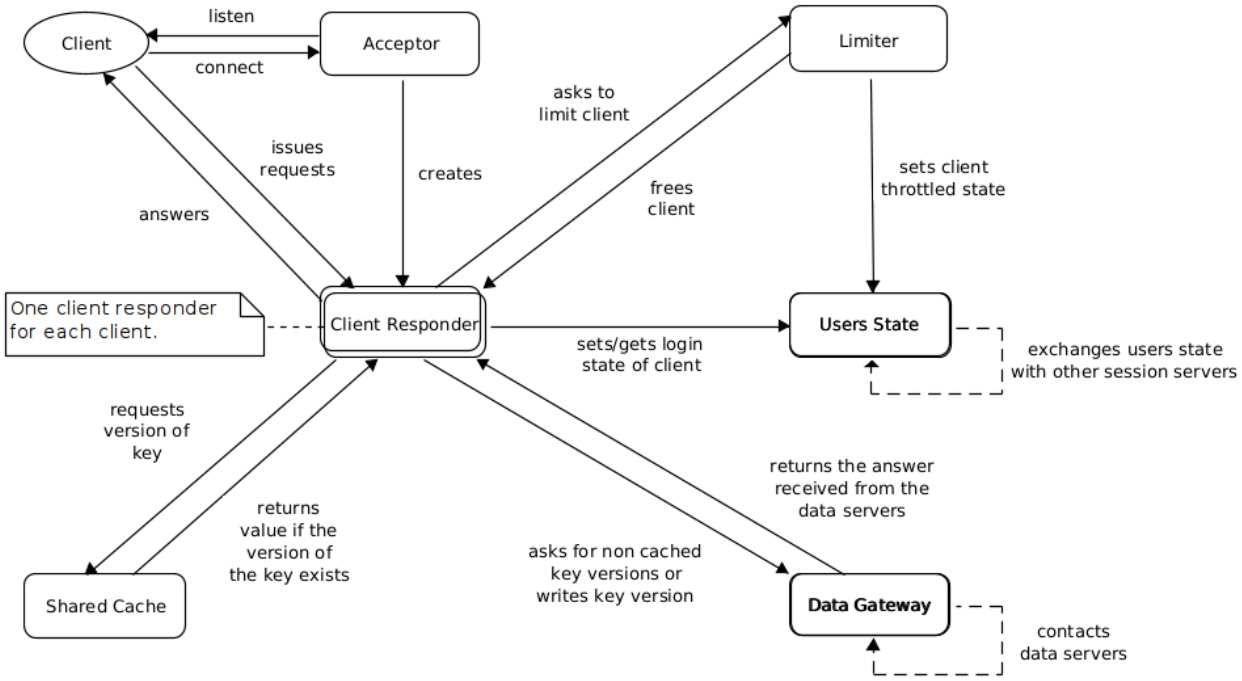


Figure 1: Arquitetura dos servidores de sessão

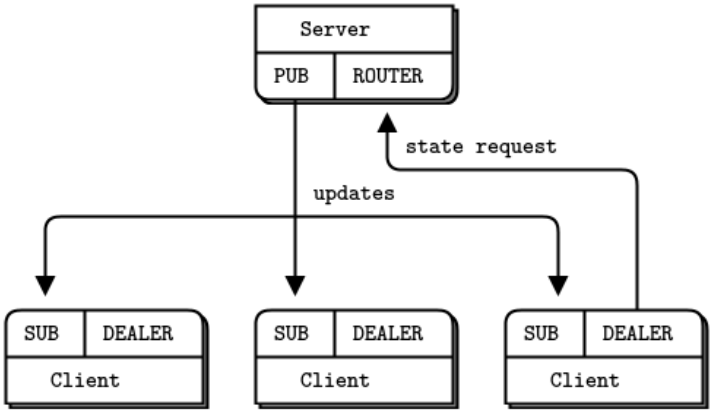


Figure 2: Replicação do estado nos servidores de sessão

4.1 Exemplos de iterações do algoritmo dos circular buffers

Para perceber os exemplos, considere a seguinte notação.

```
inc_ant = índice não circular do pedido anterior
inc = índice não circular do pedido atual

ic_ant = índice circular do pedido anterior
ic = índice circular do pedido atual

ts_ant = timestamp do pedido anterior
ts = timestamp do pedido atual
```

Outra consideração a ter é que os intervalos do *buffer* estão marcados em segundos para simplificação (na implementação é feito em milissegundos), e o timestamp que marca o ínicio do buffer é 0 (na implementação, o timestamp que marca o ínicio do buffer, é o timestamp em que o buffer foi criado). Para atualizar o circular buffer, é necessário saber o timestamp que determina o início ”global” do buffer, o timestamp do pedido anterior (quando o circular buffer é criado, este valor é igual ao timestamp de criação do buffer), e o timestamp do pedido atual. Com estes 3 timestamps conseguimos descobrir para cada um dos pedidos, qual é o índice não circular, i.e., o índice do buffer caso este tivesse tamanho infinito, e o índice circular. O índice circular corresponde ao resto da divisão inteira do índice não circular pelo número de slots. Estes índices são essenciais para calcular quais são os slots que devem ser reiniciados (cujo valor deve passar para 0), e para calcular o slot cujo valor deve ser incrementado em 1 unidade após o reinício dos valores necessários (índice circular do pedido atual).

Nos exemplos seguintes podemos observar como os slots entre o pedido anterior e o pedido atual (incluindo o do pedido atual) são reiniciados. E posteriormente incrementa-se em uma unidade o valor do slot correspondente ao pedido atual.

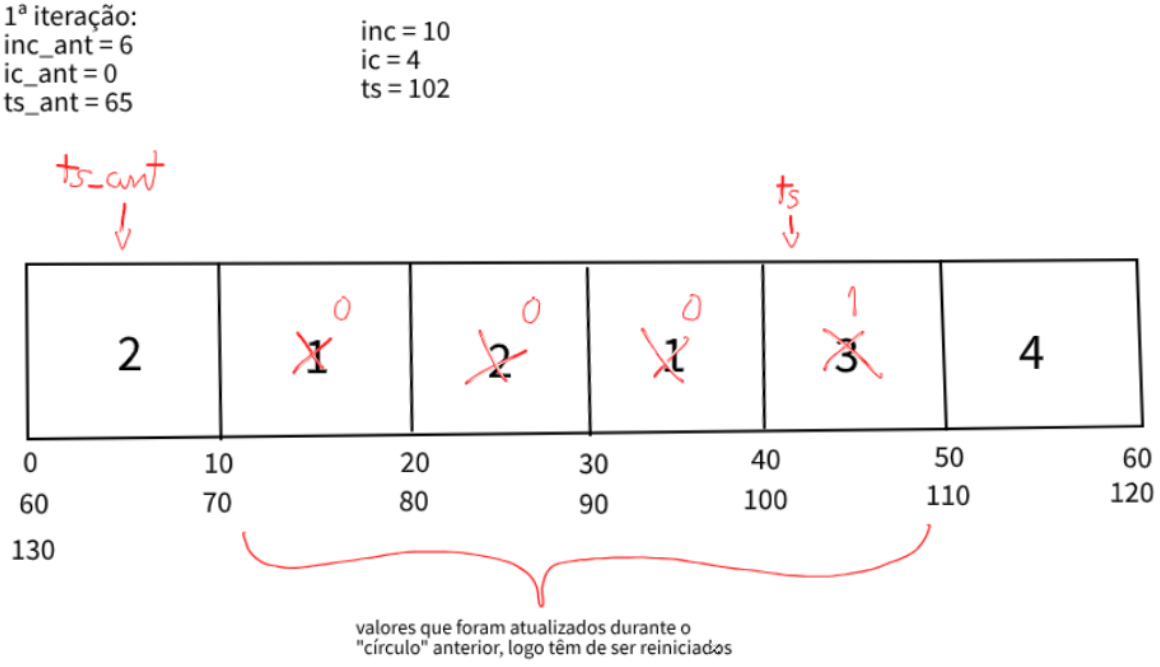


Figure 3: 1ª iteração exemplo do algoritmo de atualização dos *circular buffers*

No segundo exemplo, apresentado abaixo, é possível verificar um caso em que o buffer ”dá a volta”.

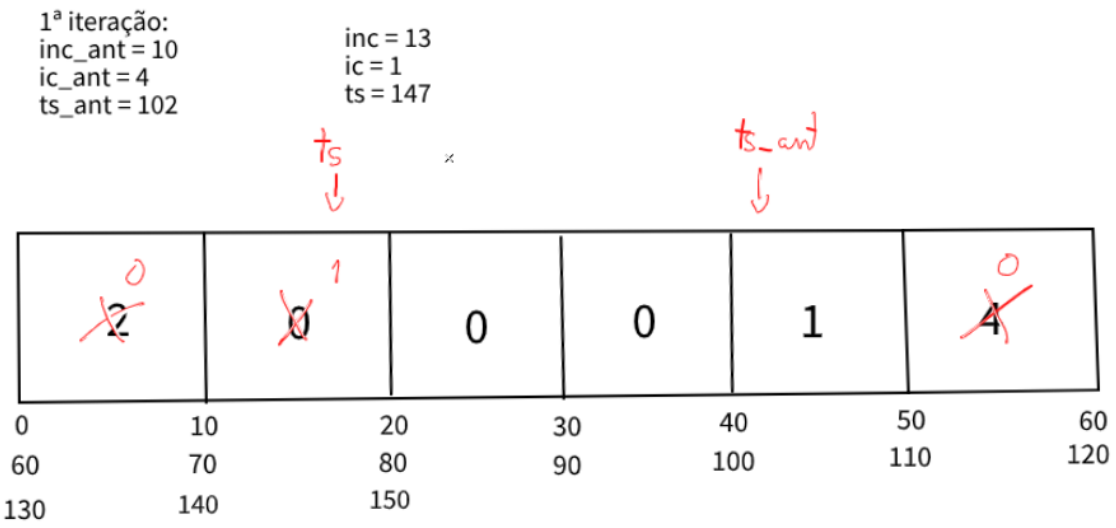


Figure 4: 2ª iteração exemplo do algoritmo de atualização dos *circular buffers*

Um caso não representado aqui, é o caso em que o índice não circular dos dois pedidos coincide. Nesse caso, apenas é incrementado o valor do slot.