Universidade do Minho
Escola de Engenharia

Fundamentos de Sistemas Distribuídos

Mestrado Integrado em Engenharia Informática

Distributed Storage System

[Trabalho Prático]

A84961 Alexandre Ferreira
A84462 Alexandre Miranda
A85227 João Azevedo
A85315 Miguel Cardoso
A85729 Paulo Araújo

Braga,
Janeiro 2021

Resumo

Este trabalho consiste na elaboração de um sistema distribuído de armazenamento em memória de pares chave-valor, segundo um modelo de programação assíncrono, muitas vezes preferível quando o objetivo acaba por ser a escalabilidade com base no número de clientes, servidores e a forma adotada para fazer a gestão de conexões e pedidos, fazendo uso de relógios lógicos. Este projeto demonstra uma implementação deste sistema utilizando a *package* Atomix, em Java, num ambiente de número de servidores fixo, assumindo que não há falhas.

Detalhe da solução

A partir desta secção iremos detalhar toda a estrutura de suporte ao sistema de armazenamento distribuído, desde o *backend* de servidores, como a interface de acesso, por parte de clientes, ao mesmo, através de um conjunto de operações fixas.

Toda a solução está, de forma simples, explicada através de exemplos, incluindo diagramas de fluxo entre os diferentes participantes, terminando com uma análise global de desempenho onde testamos diferentes configurações para os servidores com vista a perceber a escalabilidade da estrutura.

Backend (Storage Servers)

O sistema proposto implementa um conjunto de servidores fixo, indiferenciáveis, que armazenam pares chave-valor, sendo a distribuição de chaves dada através de um método de *hashing* que mapeia um dado *Long* (chave) para o *Server ID* correspondente:

```
destination_server_id = key % nr_servers //circular array style
```

Com esta relação não garantimos, obviamente, uma distribuição equitativa de chaves por N servidores se as mesmas forem aleatórias, mas visto que este não é o foco do sistema, decidimos implementar um mapeamento simples, idêntico ao verificado em *arrays* circulares, em que cada servidor armazena um conjunto disjunto de chaves ($keys\ (sv.\ X) \cap keys\ (sv.\ Y) = \emptyset$).

```
Example (w/ nr_servers = 2):
    1. Client request = PUT (key 0 to 500), then:
    2. EACH server gets 250 keys, being:
        - server ID 0: key 0, 2, 4, ... (even keys)
        - server ID 1: key 1, 3, 5, ... (odd keys)
```

Deste modo, precisamos agora de perceber como deve um servidor individual fazer a gestão dos pedidos e tirar partido de relógios lógicos na ordenação das mensagens trocadas, sendo este desenvolvido de forma modular e maximizando o código independente. Com vista a implementar isso mesmo, estabelecemos um conjunto de pressupostos na arquitetura do sistema:

1. A gestão de mensagens em programação por eventos assíncrona recorre à *package Atomix* para desenvolvimento do sistema distribuído;
2. Cada Servidor deve ser capaz de responder a um qualquer pedido de um cliente, mesmo não sendo ele o destino final de uma ou mais chaves;
3. A comunicação de pedidos entre servidores não interfere com a comunicação de pedidos de clientes;
4. Cada pedido de um cliente será interpretado como uma transação (PUT ou GET), só sendo devolvida a resposta quando todas as chaves forem processadas;
5. Cada transação PUT tem associado uma marca temporal (*timestamp*), baseada em tempo lógico, que corresponde a um *snapshot* do relógio local do servidor, usada para detetar situações de concorrência;
6. A comunicação entre servidores é feita recorrendo a uma política de *broadcast*, importante na manutenção da consistência do tempo lógico num Sistema Distribuído;

O diagrama que apresentaremos de seguida representa os dados armazenados no servidor, que corre um serviço denominado *StorageService*, que regista e implementa todos os pressupostos descritos através da enumeração anterior:

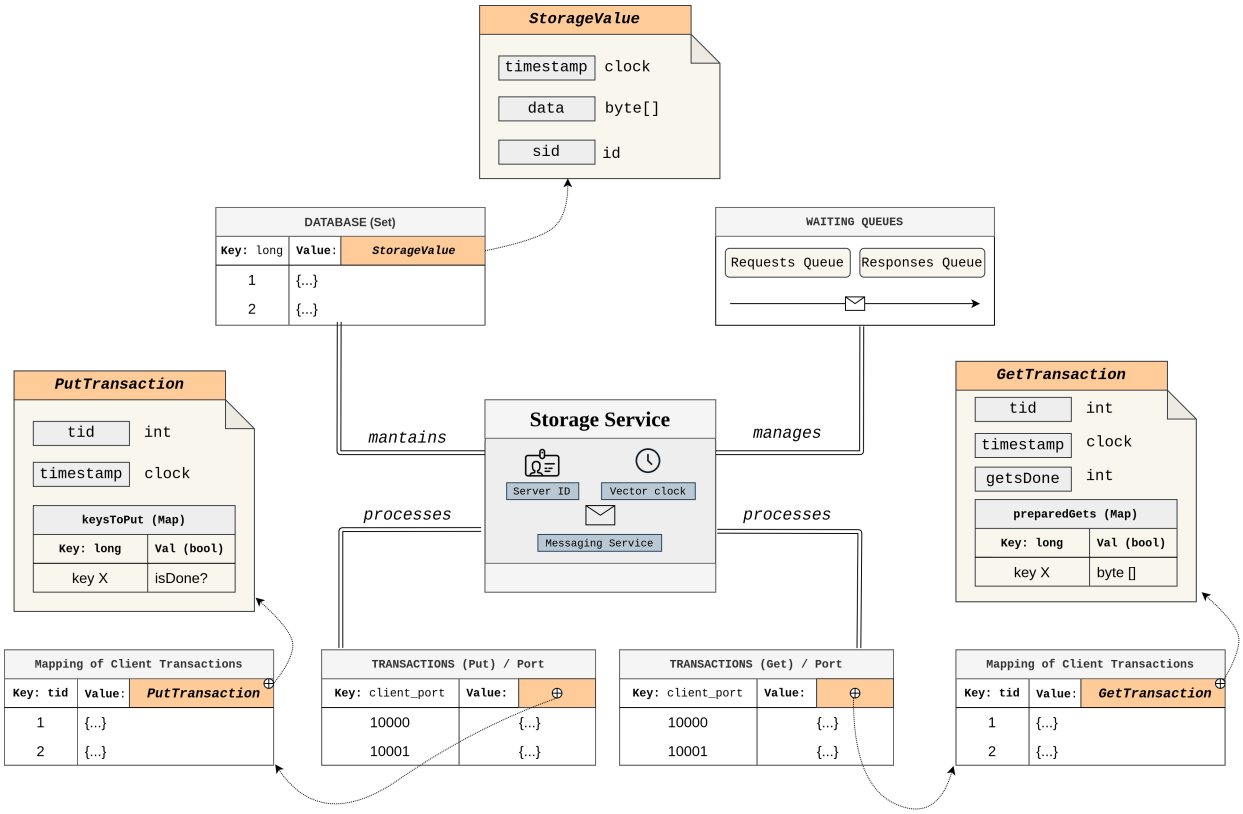


Figura 1: *Variáveis de instância de um Storage Service.*

O desenvolvimento deste servidor teve em conta que cada servidor deve, sendo o seu código respetivo bastante generalizado, responder a um qualquer pedido de um cliente, o que se encontra bastante adaptado a uma situação real onde, por exemplo, cada um destes nodos do *backend* pode representar um servidor de um país que, recursivamente, como verificado em sistemas de DNS, consulta outros para obter informações.

Em primeiro lugar, sendo este um serviço prestado por um *StorageService*, devemos perceber que tipo de eventos estão registados por este servidor[1]:

- Eventos/Mensagens com clientes:

  ```
  register_client_put/get();
  ```

- Eventos/Mensagens entre servidores:

  ```
  register_server_request/response_put/get();
  register_server_update_clock();
  ```

De notar que todas estas comunicações recorrem a objetos em memória, estabelecidos na *package app.server.data* (para objetos entre os servidores) e a *package app.data* (para objetos entre o servidor e a *StorageAPI*/cliente), recorrendo à serialização em binário como padrão de comunicação mais eficiente, compacto e flexível[2].

Devemos então perceber como diferentes pedidos são processados através de um servidor qualquer deste sistema, começando por uma transação de PUT e recorrendo a um exemplo prático:

1. Cliente faz o pedido de PUT, através da API disponibilizada, por exemplo:

[1]Todos os excertos/referências a código encontram-se desenvolvidos, originalmente, na *package app.server.handler*.
[2]Para o efeito foi desenvolvida uma *class app.Serialization* que serializa um objeto genérico.

```
future = put (tid: X, {key 1: val [...], key 2: val [...], ...})
//tid = transaction id
//can assume random server destination or fixed (explained later)
```

2. O Servidor recebe e processa um evento de `client_put`, através do *handler* especificado, `register_client_put`, e o processamento é feito da seguinte forma:

   (a) Registo de uma transação PUT com uma cópia do relógio atual do servidor:

   ```
   t = new PutTransaction (map_to_insert, copy(local_clock))
   list = getTransactions(port) //current client transactions
   add(t, list)
   process(t)
   ```

   (b) Processamento genérico de uma transação (*process(t)*):

   ```
   for each key in transaction.entries():
       destinationID = find_server(key)
       if myid == destinationID:
           //process key, check existance, conflicts
       else
           clock[ID]++
           //ask (async) destination server     (w/ clock)
           //ask (async) others to update clock (w/ clock)
   ```

   O que se torna importante reter deste algoritmo é o seguinte: caso o destino da chave que se está a analisar neste momento não for o ID do servidor que a está a processar, então torna-se necessário pedir ao destino para atualizar uma dada chave.

   Certo é que para o fazer recorremos a uma comunicação, assíncrona, com base no envio de um objeto que contém, entre outros, a chave que se pretende atualizar, o valor respetivo, o *timestamp* da transação a que corresponde essa chave e o relógio atual do servidor, previamente incrementado antes do envio.

   Em segundo lugar, é necessário enviar para todos os outros, que não o próprio servidor e o destino, ou seja, para $nr\_servers > 2$, uma mensagem que os obrigue a acompanhar o relógio dos outros servidores e, só assim, conseguiremos estabelecer, por exemplo, ordenação de mensagens para estabelecer concorrência de acessos num sistema distribuído, através de uma política de *broadcast*.

   (c) Processamento genérico de um pedido de PUT de um servidor *(process(key))*:

   ```
   if (my_database.has(key)):
       last_value = my_database.get(key)
       tstamp = last_value.timestamp()
       if (are_concurrent(tstamp, key_tstamp)):
           //resolve conflict (process with smaller id wins)
       else
           //choose biggest clock
           update_to_last_version()
   else
       my_database.put(key, (value,key_tstamp,id))
   ```

   O processo de inserção de um valor requere a verificação de conflitos e onde critérios de desempate, como o ID do processo/servidor, comparando o ID daquele que fez o pedido e o ID do que fez a última alteração ao valor de uma chave.

   A verificação de concorrência pressupõe a relação estabelecida para definir quais pedidos são concorrentes para implementações com relógios vetoriais e está definida, assim como a regra de entrega causal, na classe Java *app.server.clock.LogicalClockTool*.

3. As mensagens recebidas nos servidores são processadas cumprindo as regras de causalidade para vetores lógicos, descrita de seguida:

```
handler(message_type: "...", (source, bytes) -> {
    message = desirialize(bytes)
    if (can_be_executed(message, local_clock)):
        process(message)
    else
        enqueue(message)
})
```

Primeiramente, a condição apresentada acima verifica se o relógio fornecido na mensagem recebida pode ser lido pelo servidor, quando comparado com o seu relógio local, quer isto dizer, se cumpre a regra da entrega causal. Caso esta não se verifique, então adicionamos a mensagem à fila de mensagens global do servidor, para que mais tarde seja despachada (após processar uma outra que tenha conseguido executar, corre-se uma função que varre a fila de espera que verifica os relógios e determina se pode ou não ser processada, aplicando novamente a regra descrita acima). Assim garantimos, por exemplo, que se um servidor faz os pedidos X, Y e Z, por esta ordem, então as mensagens serão processadas pela mesma ordem.

Nota: Em anexo (8) encontra-se um diagrama geral de algumas das comunicações e considerações apresentadas ao longo destes tópicos.

Client API

Relativamente à comunicação entre o cliente e o sistema é disponibilizada uma interface fixa de operações, a partir da qual o cliente pode fazer certos pedidos e obter as respostas correspondentes. A resposta é disponibilizada sob a forma de um *Future*, ou seja, todos os pedidos são assíncronos:

1. Inserção de items: `CompletableFuture<Void> put(Map<Long,byte[]> values)`

   Para fazer uma demonstração da aplicabilidade do mesmo vamos recorrer a um exemplo onde existe concorrência de inserções explítica:

   - No código do servidor, adicionar a linha:

     ```
     if (serverID == 0):
         my_database.put(key 0, val "version1", tstamp [1,-1])
     ```

     Com isto, inserimos um valor na base de dados com um *timestamp* (impossível, apenas ilustrativo para uma situação de concorrência);
   - Iniciar dois servidores, com IDs 0 e 1 e, de seguida, fazer um pedido, a partir do cliente 0 para alteração dessa chave:

     ```
     API.put({key 0 => val "version2"})
     ```

```
09:55:27,813 [INFO] : received client 192.168.1.77:10100 PUT request (transaction: 0) | timestamp = [0,0]
09:55:27,816 [INFO] : PUT/[0] processing key 0 [NOT my key]
09:55:27,841 [WARN] : [server_request_put] PUT/ requesting server 10000 to put key 0
```

Figura 2: Pedido do Cliente 10100 (CID = 0) para o Servidor 10001 (SID = 1).

```
09:55:27,892 [INFO] : PUT/ [from: 192.168.1.77:10001] received server request put
09:55:27,893 [INFO] : [from: 10001] received server request put key 0 | [0,0]
09:55:27,893 [INFO] : [key: 0] time conflict, [WIN] NOT updating...
09:55:27,906 [INFO] : dispaching events...
```

Figura 3: Pedido do Servidor 10001 (SID = 1) para o Servidor 10000 (SID = 0).

2. Consultas: `CompletableFuture<Map<Long,byte[]> get(Collection<Long> keys)`

   Um exemplo prático para as consultas pode incluir ver o valor da chave 0, anteriormente testada para concorrência, de modo a perceber se foi atualizada:

- Fazer um pedido, a partir de um cliente 2, da chave 0, para o servidor, por exemplo, com SID = 0:

  `API.get([0])`

  Onde se obteve a seguinte resposta:

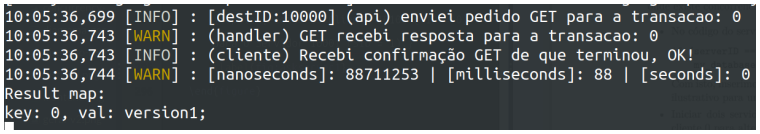


Figura 4: Consultar o valor da chave 0 ao servidor 10000 (SID = 0).

Assim, conseguimos verificar que o valor da mesma não foi atualizada pois, segundo a nossa política, havendo conflito de tempo lógico (relógios concorrentes), então, o valor que permanece corresponde ao servidor com menor ID.

Project Configuration

O projeto foi todo desenvolvido pensando numa generalização do funcionamento de um qualquer servidor, que serve pedidos de consulta e inserção, assim como uma configuração e automatização com vista a analisar a *performance*.

1. Ficheiro de configuração, no formato *TOML*,[3] que contém:
   - `"nr_servers"`: contém o número de servidores existentes no sistema
   - `"server_thread_pool_size"`: corresponde ao número de *threads* da *thread pool* do *messaging service* do servidor. De salientar que também existe um valor de *thread pool* para o cliente (não tão importante nos testes de desempenho).
   - `"init_server_port"`: representa o valor inicial a partir do qual podem ser representadas as portas dos servidores. Isto aliado ao *id* permito-nos identificar a porta especifica de cada servidor, existindo uma representação idêntica para as portas dos clientes.
2. Automatização de testes:

   Foi desenvolvido ainda um *script*[4] de inicialização do número de servidores especificado na configuração, o que facilita a eficiência na realização dos testes, visto que iniciá-los todos, um a um, ainda é uma tarefa que ocupa bastante tempo.
3. Organização de *packages*:

   As duas camadas principais Cliente e Servidor são um pouco semelhantes no que toca à sua organização. Ambas são compostas por um *data* e por um *handler*. O primeiro permite identificar os objetos que estão envolvidos nas comunicações correspondentes de cada camada, dependendo se estamos perante um *request/response* e o tipo de operação *put/get*. Já o segundo é onde se encontra toda a lógica no tratamento de mensagens, ou seja tudo o que é feito após a receção de um certo tipo de mensagens.

Performance analysis

Esta secção corresponde a uma análise de desempenho no que toca a carga que cada servidor consegue processar e o consequente *throughput* obtido para vários testes onde fazemos variar alguns parâmetros, que achamos mais impactantes na complexidade no que toca à troca de mensagens, estabelecendo o seguinte padrão de testes (para inserção de chaves):

[3] O ficheiro de configuração denomina-se *config.toml* e encontra-se na pasta *src*.
[4] O *script* denomina-se **init.sh** e encontra-se em *src/storageserver*.

1. Variar o número de chaves inseridas:

   Visto que a forma de partilha de mensagens entre servidores segue uma política de *broadcast*, então o envio de uma mensagem é feito para todos os participantes (servidores), ou melhor, envia-se o pedido para um certo destino mas todos os outros que não a origem e o destino recebem um "`update_clock`", para um número de servidores > 2. Isto permite-nos tirar conclusões em relação à complexidade do tráfego de mensagens no *backend*:

   ```
   Para X servidores e um pedido de N keys contínuas
   (que terão de ser distribuídas de forma uniforme):

   Número de pedidos a outros servidores: P = N/X
   Tráfego gerado por um servidor: T = (P*(X-1))^2 para X > 0
   ```

   O tipo de pedido realizado (put ou get) não se diferencia no número de mensagens trocadas.

2. Variar o tamanho do valor (em bytes) associado a uma chave:

   Este valor permite gerar uma carga maior em cada mensagem trocada, o que tem um impacto direto no processo de serialização dos dados.

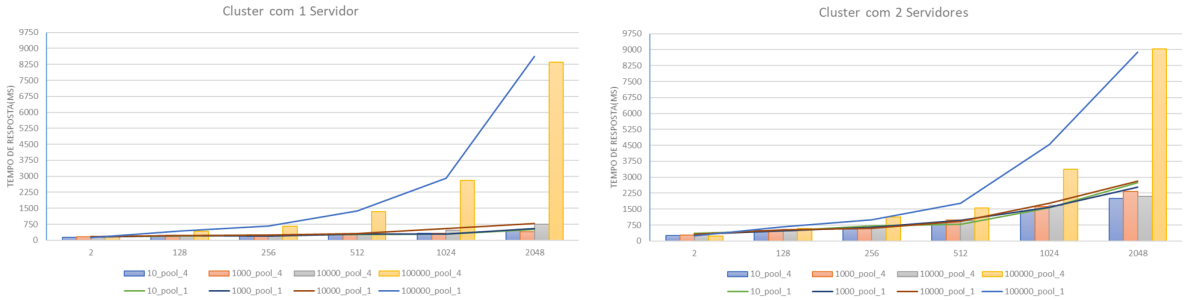
3. Variar o número de servidores (*backend*):

   Ao aumentar o número de servidores o trabalho será constantemente distribuído por todos eles e para um grande número de servidores o primeiro ponto (número de chaves) poderá ter mais impacto no desempenho.

4. Variar o tamanho da *thread pool* para os servidores:

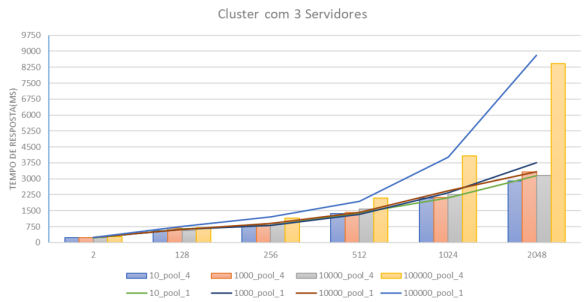
   No decorrer do desenvolvimento do código para este sistema distribuído existiu a necessidade de aplicar controlos de acesso a regiões críticas e fazer variar este parâmetro pode, por um lado, aumentar o paralelismo das operações mas também sofrer com situações de *locking*.

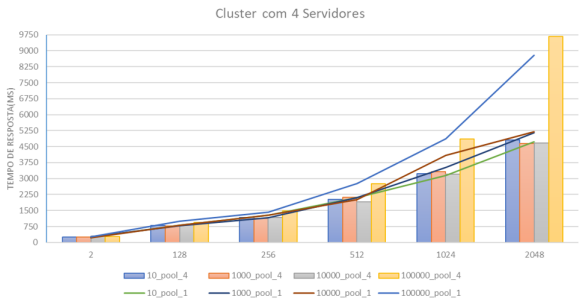
O eixo vertical de cada gráfico representa o tempo de resposta (desde a preparação do pedido até à receção da resposta) e o eixo horizontal representa o número de chaves inseridas em cada teste. As linhas nos gráficos correspondem a uma definição de *thread pool* $= 1$, enquanto que as barras representam uma *thread pool* de tamanho 4. E cada linha e barra possuem um valor diferente, que corresponde ao tamanho em bytes do valor da chave a inserir.



Em primeiro lugar para um *cluster* com um servidor percebemos que o tempo de resposta mantém-se praticamente o mesmo para valores de chave na ordem dos 10, 1000 e 10000 *bytes*, tendo um impacto muito maior, chegando até a demorar 9 segundos a responder quando consideramos chaves de 100000 *bytes*, para 2048 inserções. O tamanho da *thread pool* é algo que teve pouco impacto no desempenho de um *cluster* com apenas um servidor.

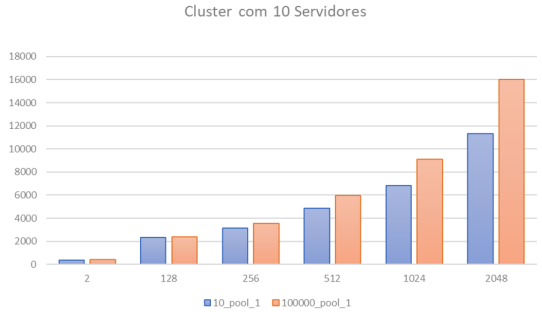
Quando temos dois servidores e como as chaves são distribuídas pelos dois existe a necessidade de fazer pedidos e receber respostas, algo que tem mais impacto nos tempos de resposta quando fazemos variar o número de chaves, ao contrário do que se verifica num *cluster* com apenas um servidor.





Em segundo lugar sabemos também que a mensagem extra de `update_clock` só existe para três ou mais servidores no *cluster* e à medida que vamos aumentando o número de servidores o desempenho global do sistema vai sendo cada vez pior. Podemos também concluir que fazer variar a *thread pool* é um fator que não é tão impactante no tempo de resposta do sistema, muito provavelmente devido ao mecanismo de controlo de concorrência adotado. Em particular podemos verificar que quanto maior é o número de servidores menor a diferença de tempos de resposta quando fazemos a ordem de grandeza do tamnho do valor da chave e, por exemplo, para quatro servidores as linhas aproximam-se mais umas das outras, assim como as barras.

Como teste final e para perceber a escalabilidade de um *cluster* de armazenamento de dados como este podemos verificar o crescimento abrupto do tempo de resposta para dois simples casos, onde são estabelecidas as diferenças entre usar 10 bytes no valor da chave ou 100000, o que nos leva a concluir todos os pontos referidos até agora, isto é, quanto mais servidores existirem menos impactante é o tamanho do valor da chave e o tráfego é tanto maior quanto o número de chaves requisitadas.



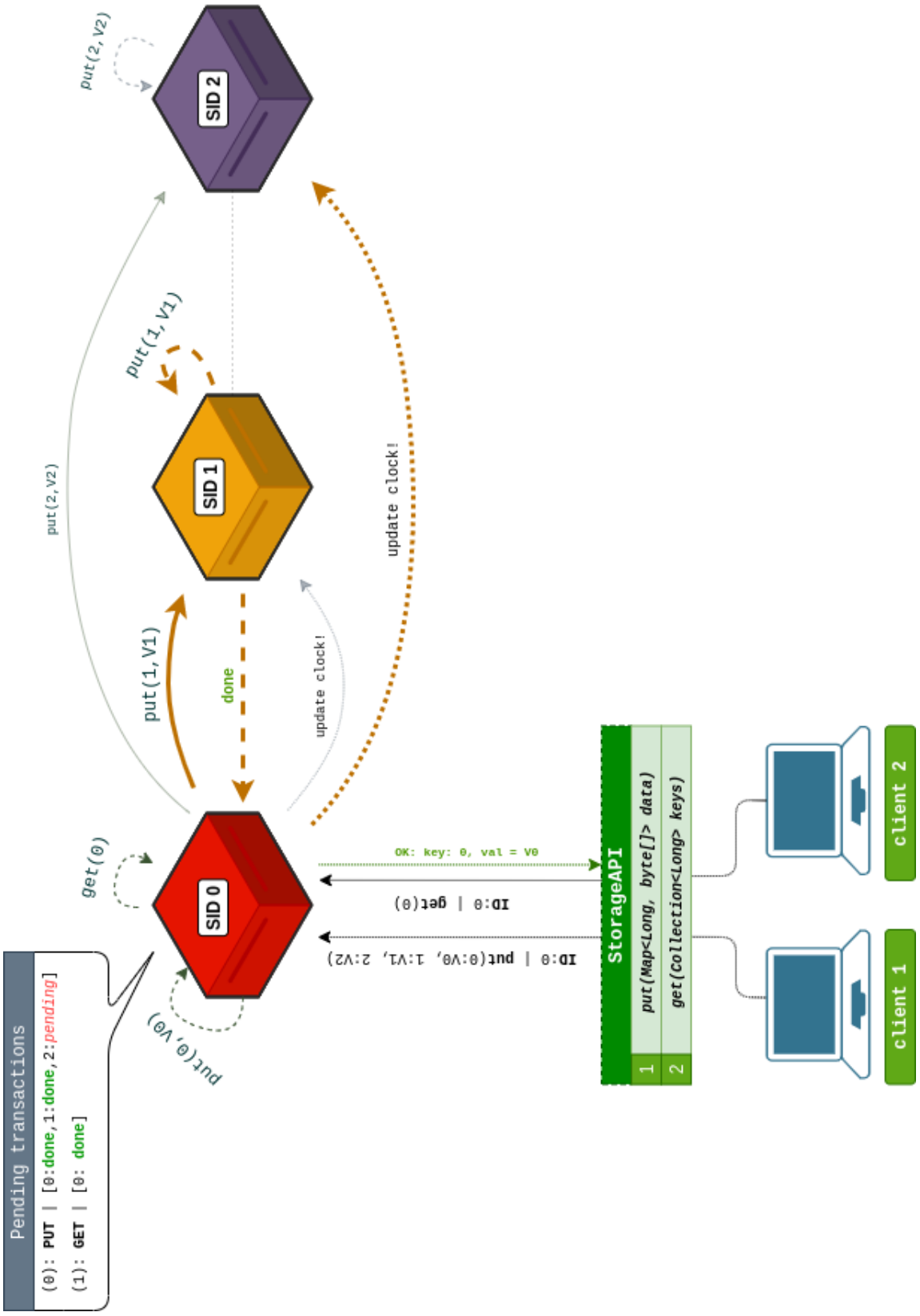
Nota: As tabelas usadas para o desenvolvimento destes gráficos encontram-se num ficheiro *excel* em *src/tests/AnalisePerformanceFSD.xlsx*

Conclusão

Em suma, fazemos um balanço positivo do trabalho desenvolvido, isto é, apesar de não termos atingido todos os objetivos estabelecidos para este TP, conseguimos criar um sistema sólido e suficientemente genérico, configurável e onde foi possível perceber os limites do mesmo e a sua escalabilidade, através de uma fase extensiva de testes, bastante importante na nossa opinião.

Relativamente às principais dificuldades, estas surgiram em grande parte devido à complexidade da sincronização dos diversos servidores e a consequente entrega causal dos pedidos, muito também por ser necessário garantir que escritas concorrentes entre clientes concorrentes apenas permitem que uma transação prevaleça, no entanto, este objetivo foi concluído.

Por fim, a valorização que corresponde a leituras concorrentes não observarem escritas parciais levou a que tentássemos, sem sucesso, diversas implementações, seguindo um pouco um analogia a *snapshot isolation* em Bases de Dados.


SID 2
SID 1
SID 0
put(2, V2)
put(1, V1)
put(2, V2)
put(1, V1)
done
update clock!
update clock!
get(0)
put(0, V0)
OK: key: 0, val = V0
ID:0 | get(0)
ID:0 | put(0:V0, 1:V1, 2:V2)
StorageAPI
put(Map<Long, byte[]> data)
get(Collection<Long> keys)
1
2
client 1
client 2
Pending transactions
(0): PUT | [0:done,1:done,2:pending]
(1): GET | [0: done]

A. Definir parâmetros de configuração no ficheiro config.toml (exemplo):

```
[defaults]
nr_servers = 2
server_thread_pool_size = 1
...
```

B. A execução do backend pode ser feita de duas formas:

```
$ cd src/storageserver
```

1. Script de inicialização:

```
//utiliza a config para inicializar N servidores
$ bash init.sh
```

2. Manualmente:

```
//servidor 10000 + 1 = 10001
$ make run_storageserver
arguments: 1
```

B. A execução do cliente recorre de uma main (Client.java) que faz uso da API (StorageAPI.java):

Definir o caso de teste na main:

```
//inicializar mapa para enviar no put
...

//definir id servidor destino (pode ser aleatório)
API.setDestinationID(1);

CompletableFuture<Void> result = API.put(...);
result.thenAccept(r -> {
    //processar resposta
})
```

Nota: Já se econtram definidos testes para diferentes IDs de clientes e, portanto, podem ser seguidos os exemplos lá descritos.