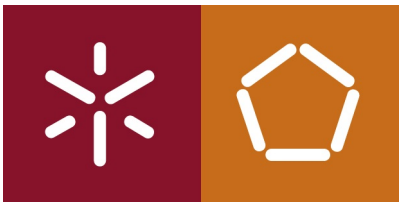

Universidade do Minho

Mestrado Integrado em Engenharia Informática

Fundamentos de Sistemas Distribuídos

Armazenamento Persistente de Pares Chave–valor

Ana Rodrigues	(A78763)
Armando Santos	(A77628)
Cláudia Correia	(A77431)

4 de Janeiro de 2019

Conteúdo

Resumo

Este documento diz respeito ao trabalho prático proposto na unidade curricular de Fundamentos de Sistemas Distribuídos da Universidade do Minho. O objetivo deste projeto consiste no desenvolvimento de um sistema distribuído de armazenamento persistente de pares chave-valor, utilizando Java e Atomix.

1 Introdução

A elaboração do presente documento tem como finalidade justificar as estratégias adotadas na criação de um sistema distribuído de armazenamento persistente de pares chave-valor, utilizando Java e Atomix. O sistema deverá utilizar vários servidores para armazenar de forma persistente chaves (`Long`) a que são associados valores (`byte[]`) oferecendo as seguintes operações aos clientes:

- `CompletableFuture<Boolean> put(Map<Long,byte[]> values)`: Escreve um conjunto de pares chave–valor, indicando se teve sucesso.
- `CompletableFuture<Map<Long,byte[]>> get(Collection<Long> keys)`: Lê os valores associados a um conjunto de chaves.

Para além disto, o sistema deverá garantir também que, ou todos os pares chave–valor incluídos no pedido de escrita são modificados, ou então nenhum é modificado, mesmo que haja falha e reinicio de alguns servidores. No entanto não se colocam quaisquer restrições ao que leitores concorrentes observam. Admite-se que a operação de leitura bloqueie ou devolva uma exceção em caso de falha e reinicio dos servidores.

De forma a garantir o correto funcionamento do sistema, serão também implementadas funcionalidades adicionais que serão descritas ao longo das secções seguintes.

2 Estrutura do sistema

Como ponto de partida para a resolução do problema, foram tomadas diversas decisões quanto à estrutura da solução, de modo a facilitar a implementação da mesma e a cumprir com os requisitos mínimos obrigatórios. Tendo isto em conta, foram definidos os componentes que contribuem para o funcionamento do sistema, assim como a forma como estes deverão comunicar entre si.

2.1 Componentes

Primeiramente foram identificados os intervenientes do sistema - *Client*, *Server*, *Forwarder* e *Manager* -, tendo cada um deles uma função a desempenhar:

- ***Client*:** comunica ao *Forwarder* a intenção de efetuar uma operação de *get* ou *put*;
- ***Server*:** armazena um determinado domínio de pares chave-valor;
- ***Forwarder*:** conhece a topologia da rede (quais os *Servers* participantes e que pares chave-valor cada um armazena) e encaminha para cada *Server* os pares chave-valor que lhe são destinados;
- ***Manager*:** responsável pelo *2 Phase-Commit*, garantindo que os vários servidores participantes concordam numa decisão.

De maneira a que a aplicação seja adequada à utilização em grande escala, decidiu-se permitir a customização da arquitetura de suporte à aplicação. Os papéis de *Forwarder* e *Manager* deverão ser modulares, permitindo que qualquer um dos *Servers* os instancie.

2.2 Funcionamento geral

No que diz respeito ao *Client*, este deverá permitir as duas operações de (*put* e *get*) que, ao serem requisitadas, serão tratadas pelo *Forwarder*. Este será, então, responsável por receber os pedidos dos *Clients* e encaminhá-los para os *Servers* participantes. Para tal ser possível, o *Forwarder* deve conter toda a informação acerca dos servidores participantes no sistema, evitando, assim, que estes últimos necessitem de se conhecer entre si e que guardem informação repetida. Mediante o tipo de operação a efetuar, o procedimento levado a cabo pelo *Forwarder* deverá diferir em alguns aspetos, tal como veremos de seguida.

Por sua vez, o *Server* necessita de armazenar os pares chave-valor que lhe pertencem, intervir no *2 Phase-Commit* e responder ao *Forwarder* com os resultados das operações que lhe forem solicitadas.

No que toca ao *Manager*, este terá apenas o papel de coordenar o *2 Phase-Commit*, controlando assim as diferentes transações que são efetuadas.

2.2.1 Operação de put

Caso a operação seja um *put*, o *Forwarder*, que deve conhecer o *Manager*, inscreverá os participantes da transação para que se dê início ao *2 Phase-Commit*. Por fim, o *Forwarder* ficará a aguardar resultados dos *Servers* e, quando obtiver resposta de todos os intervenientes de uma determinada operação, deverá notificar o *Client* que desencadeou o processo.

2.2.2 Operação de get

Para as operações de *get* o processo será semelhante ao *put*, no entanto, os *Servers* participantes não necessitam de *2-Phase-Commit*, uma vez que não é obrigatório existir coordenação entre os diferentes intervenientes.

Com o objetivo de retratar o funcionamento geral do sistema foi criada a ilustração abaixo. A figura 1 representa uma operação de *put* ou *get*, na qual os participantes são o *Server 2* e o *Server 4*.

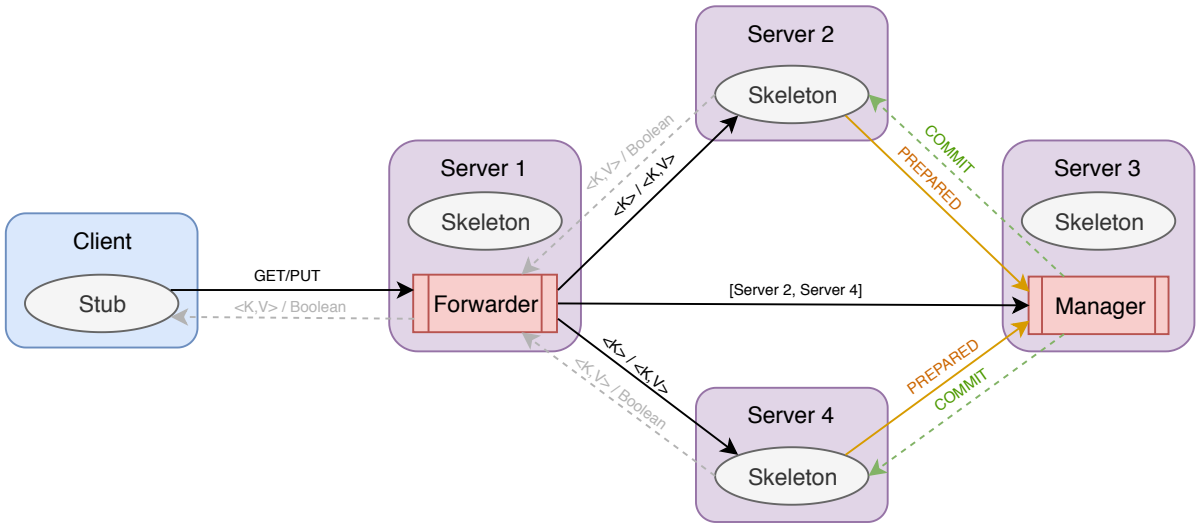

Figura 1: Funcionamento geral do sistema.

Qualquer *Server* pode desempenhar os papeis de *Forwarder* e *Manager*, inclusive, se necessário, o mesmo *Server* pode desempenhar os dois papeis. Apesar disto, em cada momento, podem existir apenas um *Forwarder* e um *Manager* no sistema.

2.3 Distribuição de pares chave-valor

Uma vez que cada *Server* deverá possuir um identificador, a distribuição de pares chave-valor deve ser feita com base nesse identificador. Isto é, ao chegar uma chave (*get*) ou um par chave-valor (*put*) ao *Forwarder*, este deverá ser capaz de distribuir estes elementos pelos *Servers* participantes com base na seguinte fórmula:

```
ID do Server = chave % número total de Servers
```

3 Implementação

Nesta secção serão descritos, de forma sucinta, alguns dos pormenores mais relevantes relacionados com a implementação do sistema. Tal como foi dito anteriormente, este projeto foi realizado com recurso a Java e à biblioteca Atomix. Posto isto, nas secções seguintes será utilizado vocabulário referente a estas duas ferramentas.

3.1 Client

O *Client* comunica com o sistema através de um *stub*. Posto isto, o *stub* necessita de indicar ao *Forwarder* qual a operação pretendida pelo utilizador e quais os pares chave-valor a esta associados. Após isto, o *stub* deverá ficar a aguardar que o *Forwarder* o informe acerca do identificador da operação, dado que cada operação de *get* ou *put* possui um identificador. De seguida, o *stub* deverá armazenar o identificador da operação, juntamente com o `CompletableFuture` a ela associado. Sendo assim, o *stub* necessita de manter dois mapas, um para os pedidos de *get* e outro para os pedidos de *put*, que ainda não foram concluídos:

```
private Map<Integer, CompletableFuture<Boolean>> putRequests;
private Map<Integer, CompletableFuture<Map<Long, byte[]>>> getRequests;
```

No momento em que o pedido de *get* ou *put* é feito, um `CompletableFuture` é retornado ao utilizador. Assim que o *Client* receber do *Forwarder* o resultado de uma operação, completa o `CompletableFuture` a ela associado, e remove a operação do mapa.

Em suma, a interação existente entre o *Client* e o *Forwarder* é a apresentada na figura abaixo.

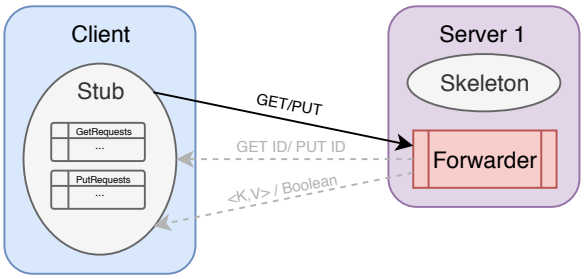

Figura 2: Interação entre o *Client* e o *Forwarder*.

3.2 Forwarder

Para armazenar os pares chave-valor associados a cada operação, assim como qual o cliente que a solicitou, o *Forwarder* necessita de estruturas específicas. Sendo assim, à semelhança do *Client*, o *Forwarder* utiliza também dois mapas:

```
private Map<Integer, PutRequest> putRequests;
private Map<Integer, GetRequest> getRequests;
```

Tal como podemos ver, o *Forwarder* mapeia o identificador de cada operação com o respetivo pedido. O `GetRequest` e o `PutRequest` são classes Java criadas para armazenar a informação relativa a uma dada operação, nomeadamente:

```
private int transactionId;
private String clientAddress;
private Map<String, Map<Long, byte[]>> participants;
private Map<String, Collection<Long>> requestedKeys;
private boolean completed;
```

Para cada `GetRequest`, para além das chaves associadas ao pedido, é também mantido registo do seu identificador, do cliente que o requisitou, dos *Servers* participantes e respetivas respostas, e de um booleano que indica se o pedido já obteve resposta ou não.

```
private int transactionId;
private String clientAddress;
private Map<String, Integer> participants;
private Map<String, Map<Long, byte[]>> keysToPut;
private boolean sent;
private boolean completed;
```

No caso do `PutRequest`, para além dos elementos utilizados no `GetRequest`, é também utilizado o booleano `sent` que indica se o *Forwarder* já enviou os pares chave-valor para todos os participantes.

3.2.1 Persistência de dados e recuperação em caso de falha

É importante saber se um pedido já foi concluído (booleano `completed`), pois, no caso do *Forwarder* falhar, este deve conseguir perceber se já respondeu àquele pedido ou não. No entanto, para que isto seja possível, é necessário também utilizar um *log* no qual o *Forwarder* guarda os pedidos (`GetRequest` e `PutRequest`) que lhe foram feitos. Quando o *Forwarder* recupera de uma falha, valida o *log*, e reconstrói os seus mapas. Ao validar o *log*, se encontrar uma operação que ainda não foi concluída, o *Forwarder* procede de forma diferente mediante a operação:

- ***get*:** inicia uma nova operação, enviando de novo as chaves para os participantes;
- ***put*:** comunica com o *Manager* para saber qual o estado da transação. Caso a transação tenha sido *committed*, significa que a resposta ao *put* foi *true*, caso tenha sido *aborted* significa que a resposta foi *false*. Se a transação ainda estiver a decorrer, o *Forwarder* continua a aguardar a resposta dos *Servers*.

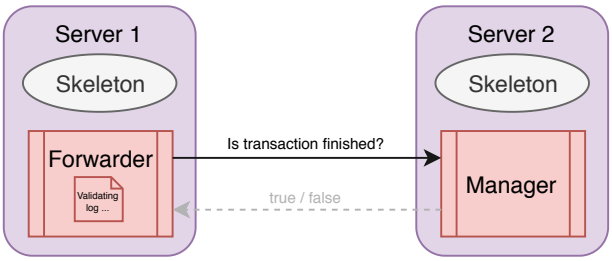

Figura 3: Interação entre o *Forwarder* e o *Manager* quando está a ser feita a validação do *log*.

É importante compreender que, caso o *Forwarder* falhe durante uma operação de *put*, esta não deve ser realizada novamente por dois motivos. Primeiro, pode levar a que valores de transações mais antigas sobreponham valores recentes e, segundo, pode fazer com que estejam constantemente a ser iniciadas transações novas, caso o *Forwarder* esteja constantemente a falhar. Por estas razões, o procedimento em caso de falha para o *put* é diferente do *get*.

Uma vez que é o *Forwarder* que inscreve os participantes nas transações, outro caso que pode ser gerado pela falha do *Forwarder* é ser iniciada uma transação sem os participantes serem notificados de tal. Para prevenir estes casos utiliza-se o booleano `sent`. Ao recuperar de uma falha, caso este booleano esteja a *false*, o *Forwarder* irá enviar os pares chave-valor de novo aos *Servers* participantes, evitando assim que as transações sejam abortadas por culpa do *Forwarder*.

Dado que cada operação de *get* possui um identificador diferente, definido pelo *Forwarder*, os identificadores que vão sendo atribuídos são também inseridos no *log*. Isto permite que, caso o *Forwarder* falhe, não sejam atribuídos identificadores iguais a diferentes operações de *get*.

3.2.2 Gestão de pedidos

No caso dos pedidos de *get*, não existe interação com o *Manager*, logo é o *Forwarder* que gere os identificadores dos pedidos. No que toca à operações de *put*, estas são controladas pelo *Manager*, sendo assim é este que cria os identificadores.

Para obter o identificador de um *put*, o *Forwarder* deve indicar ao *Manager* o início de uma transação, juntamente com os participantes da mesma. Ao ser notificado da situação, o *Manager* devolve ao *Forwarder* o contexto (identificador) da transação e inicia a transação.

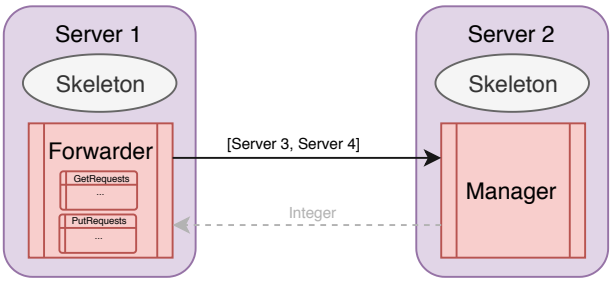

Figura 4: Interação entre o *Forwarder* e o *Manager* quando está a ser iniciada uma transação.

3.2.3 Cálculo dos participantes

Quando é requisitada uma operação ao *Forwarder*, este determina quais os *Servers* participantes com base na fórmula especificada em 2.3. Posteriormente, envia a cada um dos *Servers* os pares chave-valor (ou chaves) a si destinados. À medida que os *Servers* vão respondendo, o *Forwarder* vai armazenando no mapa `participants`, a resposta de cada um dos participantes. O resultado final da transação é devolvido ao *Client* apenas após todos os participantes terem dado a sua resposta.

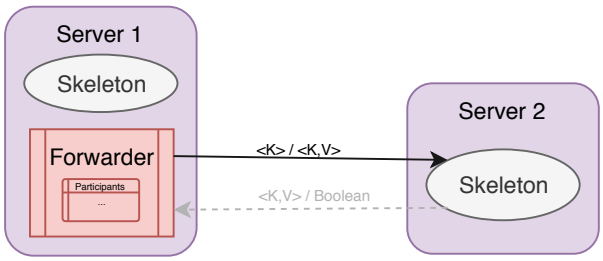

Figura 5: Interação entre o *Forwarder* e o *Server*.

3.3 Server

O *Server* é responsável por armazenar um domínio dos pares chave-valor, sendo que para isso utiliza um mapa `pairs`. No entanto, enquanto estão a ocorrer as transações, é necessário armazenar os dados de forma volátil, para o caso de a transação ser abortada. Para cada transação é, então, armazenada uma estrutura `pairsVolatile` e, apenas quando a transação é *committed*, é que os pares resultantes dessa transação são adicionados à estrutura principal `pairs`. Posto isto, os mapas utilizados pelo *Server* são os seguintes:

```
private Map<Long, byte[]> pairs;
private Map<Integer, Map<Long, byte[]>> pairsVolatile;
```

3.3.1 Persistência de dados e recuperação em caso de falha

De forma a garantir a persistência dos dados e o correto funcionamento do *2-Phase-Commit*, o *Server* escreve informação para um *log*. Nesse *log* são armazenadas entradas denominadas `LogEntry`, nas quais consta informação tanto acerca do estado da transação (*Initialized*, *Prepared*, *Committed* ou *Aborted*) como quanto aos pares chave-valor associados a essa transação. Em caso de falha, o *Server* percorre o *log* e determina quais as `LongEntry` que foram *Completed* e, para cada uma destas entradas, carrega os pares chave-valor associados para a estrutura `pairs`.

Dado que cada transação possui um identificador diferente, definido pelo *Manager*, os identificadores que vão sendo atribuídos são também inseridos no *log*. Isto permite que, caso o *Manager* falhe, não sejam atribuídos identificadores iguais a diferentes transações.

3.3.2 2-Phase-Commit

Relativamente ao *2-Phase-Commit*, como é o *Forwarder* que inscreve os *Servers* na transação, estes, após inserirem na estrutura volátil os pares chave-valor, notificam o *Manager* de que estão *Prepared*, e inserem a nova `LogEntry` no *log*. Assim que o *Manager* envia a resposta de *Commit* ou *Abort*, o *Server* armazena uma nova `LogEntry` no *log* a assinalar o término da transação.

Tal como acontece no *2-Phase-Commit*, em caso de falha, se uma transação estiver assinalada como *Initialized* no *log*, então é enviado *Abort* para o *Manager*. Caso no *log* conste *Prepared*, o *Server* envia novamente o *Prepared* para o *Manager*. Nas situações em que a transação está marcada no *log* como *Aborted* ou *Committed*, essa transação é ignorada.

3.4 Manager

O *Manager* mantém um *log* essencial para o sucesso do *2 Phase-Commit*, uma vez que, serão nele registadas todas as informações relacionadas com as transações em curso e quais os seus participantes. Para além disto, tal como acontece no *2-Phase-Commit*, em caso de falha, para as transações que estão presentes no *log* como *Initialized*, o *Manager* pergunta aos participantes se estes estão *Prepared*. Para as transações que estão assinaladas como *Committed*, o *Manager* reenvia o *Commit* aos participantes.

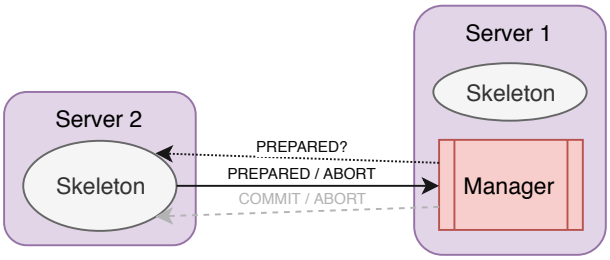

Figura 6: Interação entre o *Server* e o *Manager*.

3.4.1 Recuperação das operações de put

Tal como foi mencionado na secção 3.2, quando o *Forwarder* falha, caso encontre uma transação de *put* incompleta no *log*, este vai questionar o *Manager* acerca desta transação (figura 3). O *Manager* ao receber esta questão vai à sua estrutura verificar se a transação já foi terminada. Se a transação já tiver sido *Committed*/*Aborted* o *Manager* vai responder com *true*/*false* ao *Forwarder*, caso contrário vai enviar a questão "*Prepared?*" para os participantes. Esta operação não é bloqueante dado que a troca de mensagens, relativas a este cenário, entre o *Forwarder* e o *Manager* é assíncrona.

3.4.2 Persistência de dados

No que toca ao *Manager*, a persistência de dados é também garantida uma vez que em cada `LogEntry` são guardados os participantes da transação.

Um aspeto importante a mencionar é o facto da persistência de dados permitir que, a qualquer momento, se um *Server* que falhou antes de receber o *Commit*/*Abort* reenviar o *Prepared*, este vai receber a resposta que necessita, independentemente do *Manager* ter falhado entretanto ou não.

3.4.3 Controlo de transações não bloqueante

Caso algum dos participantes, por algum motivo, não responda *Prepared* ou *Abort* ao *Manager*, este não irá bloquear. Para este efeito foi utilizada uma `TimerTask`, denominada `AbortTask`, que é executada passado um *timeout*. Como neste caso o *timeout* corresponde a 10 segundos, se passados os 10 segundos nem todos os *Servers* tiverem respondido à transação, a `AbortTask` é lançada e aborta automaticamente a transação, notificando todos os participantes.

As `AbortTask` relativas a cada transação são guardadas num mapa e, caso todos os participantes de uma transação respondam atempadamente, o lançamento da *AbortTask* dessa transação é cancelado.

```
private Map<Integer, Timer> transactionsTimer;
```

3.5 Controlo de concorrência

O sistema foi implementado de forma a que possam ser utilizadas várias *threads* na mesma *pool*. Isto deve-se ao facto de ter sido feito controlo de concorrência, utilizando `synchronized`, nos pontos críticos da implementação. Um exemplo disto foi a utilização do mapa `transactionsState` no *Manager*. Como pode acontecer de ser atingido o *timeout* de 10 segundos no momento exato em que o último participante responde, foi necessário manter registo do estado de uma transação através de um mapa:

```
private Map<Integer, Boolean> transactionsState;
```

Este mapa está sujeito a controlo de concorrência e, apenas caso o estado da transação esteja a *false* é que a transação pode ser *Committed* ou *Aborted*. Isto impede que existam duas *threads* a fazer *abort* e *commit* da mesma transação, prevenindo assim que uns participantes recebam a mensagem de *commit* e outros a de *abort*, o que iria gerar inconsistências.

No que toca a controlo de concorrência para uma *pool multithreaded*, foi, por exemplo, utilizado `synchronized` ao incrementar o identificador de transações, prevenindo, assim, que, ao receber duas mensagens de início de transação, duas *threads* utilizem o mesmo identificador para as transações que estão a tratar.

3.6 Escritas concorrentes

Outra das garantias que se decidiu implementar foi que, no caso de duas escritas concorrentes nas mesmas chaves, mesmo que armazenadas em *Servers* distintos, os valores escritos em último lugar nas diferentes chaves provinham da mesma operação. Para isto foi necessário controlar o acesso concorrente de diferentes *threads* ao *Messaging Service* do *Manager*, e à estrutura `pairs` principal de cada *Server*.

Num dado momento, pode acontecer de duas *threads* do *Manager* decidirem dar *commit* a duas transações diferentes. Tome-se como exemplo o caso em que a *thread 1* começa a enviar as mensagens de *commit* aos participantes da transação 1, se não for feito controlo de concorrência no *Messaging Service*, pode acontecer de no mesmo exato momento existir uma *thread 2* a enviar as mensagens de *commit* da transação 2. Isto poderia fazer com que uns *Servers* recebessem primeiro a resposta da transação 1, e armazenavam os valores dessa transação na estrutura, enquanto outros estavam a armazenar os valores relativos à transação 2. Se nestas duas transações existissem pares com a mesma chave, o pressuposto mencionado acima iria ser desrespeitado. Ao aplicar `synchronized` no *Messaging Service* sempre que é feito *commit*, isto vai fazer com que todos os *Servers* recebam primeiro as mensagens de *commit* da transação 1, e só depois as da transação 2, uma vez que as mensagens são enviadas a partir de um canal FIFO.

```
synchronized (this.ms) {
    for (Address a : this.participants.get(transactionId).keySet()) {
        ...
        ms.sendAsync(a, "Manager-commit", this.s.encode(msg));
    }
}
```

No que toca ao *Server*, embora este receba primeiro as mensagens da transação 1, e só depois da transação 2, as *threads* que recebem essas mensagens podem estar a tentar inserir os dados na estrutura `pairs` em simultâneo. Se isto acontecer, não existe garantia de que os valores da transação 1 são inseridos na estrutura primeiro que os da transação 2. Para prevenir estes casos, optou-se por utilizar `synchronized` na estrutura `pairs`, de forma a que, para todos os *Servers*, sejam sempre primeiro armazenados os valores da primeira transação que recebem (transação 1), e só depois a da segunda transação (transação 2).

```
synchronized(this.pairs){
    ...
    this.pairs.putAll(keysToPut);
    ...
}
```

Ao adotar esta abordagem, independentemente das duas transações possuírem chaves em comum ou não, todos os *Servers* irão estar em concordância relativamente aos últimos valores inseridos na estrutura.

3.7 Resolução Modular

Cada um dos componentes do sistema - *Client*, *Forwarder*, *Server* e *Manager* - foi implementado de forma totalmente independente dos restantes. Tanto o *Manager* como o *Forwarder* são *threads* que podem ser lançadas por qualquer máquina, sendo que utilizam o endereço dessa mesma máquina. Ao longo deste relatório, assumiu-se que seriam *Servers* a desempenhar os papeis de *Manager* e *Forwarder*, por conseguinte estes herdaram o endereço do *Server* onde estavam a correr. No entanto, a qualquer momento o *Forwarder* e o *Manager* podem correr noutra máquina, desde que lhes seja atribuído um endereço.

É importante mencionar também que os componentes foram implementados de forma a abstrair ao máximo o contexto no qual estão inseridos, sendo que podem ser utilizados noutros cenários.

4 Conclusões e Trabalho Futuro

De uma forma geral, conclui-se que o resultado obtido cumpre com os requisitos estipulados em 1. Para além das funcionalidades básicas, foi efetuado controlo de concorrência para os casos em que são utilizados componentes *multithreaded*, foi feita uma implementação modular de cada componente, e foram prevenidas situações de inconsistência provocadas por escritas concorrentes.

Apesar disto, existem aspetos que poderiam ser melhorados, nomeadamente permitir ao utilizador customizar a topologia da rede através de um ficheiro de configuração. Em adição a isto, seria também vantajosa a utilização de mais do que um *Manager* e *Forwarder* em simultâneo, de forma a beneficiar o desempenho das operações.

Como trabalho futuro, mediante o contexto no qual o sistema distribuído de armazenamento estivesse inserido, poderia ser importante garantir a ordem pela qual as operações eram efetuadas para cada utilizador. Por exemplo, caso um utilizador efetua-se um *put* seguido de um *get*, as operações deveriam ser executadas por esta ordem, evitando que pudessem ser obtidos resultados inconsistentes.