Distributed Systems and Algorithms
2016/2016
Work Assignment TP2

A Key-Value Store using Two Different Consistency Models: PAXOS based vs. DYNAMO based

David Gago, Paulo Faria, Rui Sanches
{d.gago, p.faria, r.sanches}@campus.fct.unl.pt

Abstract

In this work assignment we implemented a Distributed Key Value Store, supported by two different approaches: exploring strong consistency model based on PAXOS and an eventual consistency model inspired by Dynamo. We developed the system using Akka – a toolkit and runtime for building highly concurrent, distributed and resilient message-driven systems, programming the system in JAVA/SCALA. We designed our system and implemented i, in a way to support both MULTIPAXOS and Dynamo, clients and servers, so the system can run both models simultaneously. We also designed the system so that the clients communicate with a middle server, and this server communicates with the rest of the servers. We conducted an experimental evaluation of the implemented system to support a comparative analysis between the two provided supports. The obtained results show that our implementation of Dynamo is faster than Multipaxos, although not by much.

1. Introduction (~half A4 page)

Este trabalho surgiu no seguimento do estudo do tema da consistência, tentando que fossem explorados dois tipos diferentes de consistência, ao implementarmos um sistema de armazenamento Chave-Valor. O objetivo deste trabalho consiste em elaborar duas versões de sistemas de armazenamento, garantindo que uma delas segue o modelo de consistência forte[4] e a outra segue um modelo de consistência eventual[4].

Ao estudar estes dois tipos de consistências foi possível verificarmos a existência de problemas associados a ambos. No caso da consistência forte, os problemas estão associados à necessidade de contactar um quórum de réplicas para executar uma operação, sendo que isso levanta a questão da eficiência do sistema contra a sua consistência, visto que as réplicas podem por exemplo estar localizadas em pontos geográficos distantes, introduzindo uma maior latência.

No caso da consistência eventual, esta solução permite uma maior flexibilidade ao sistema, evitando que este seja tão centralizado, permitindo ainda que, operações sejam executadas contactando-se apenas um número mais reduzido de réplicas, quando comparado com a solução da consistência forte. No entanto esta solução também apresenta problemas, sendo um destes o problema da reconciliação de divergências entre estados de réplicas diferentes.

Para testar esta dualidade, implementou-se um modelo baseado no Paxos[3](versão implementada Multipaxos), que representa a consistência forte e no Dynamo[5], que representa a consistência eventual. Estas implementações apresentaram alguns desafios, sendo um deles por exemplo garantir que os clientes obtêm sempre resposta a

qualquer operação que façam, quer esta seja uma leitura ou escrita, mesmo em situações de falha ou adição de servidores.

A nossa implementação de maneira geral atingiu os objetivos propostos, permitindo que façamos uma avaliação dos modelos de consistência, em termos de operações por segundo, usando vários tipos de condições diferentes, como a variação do número de servidores ou mesmo a variação do número de vezes que uma chave é replicada. No entanto a nossa solução contém um problema, relativamente a estas variações para o caso do Multipaxos, que não é garantido que funcione sempre. No caso do Dynamo funciona sem qualquer problema, suportando estas variações dinâmicas.

2. System design

2.1 System model and architecture

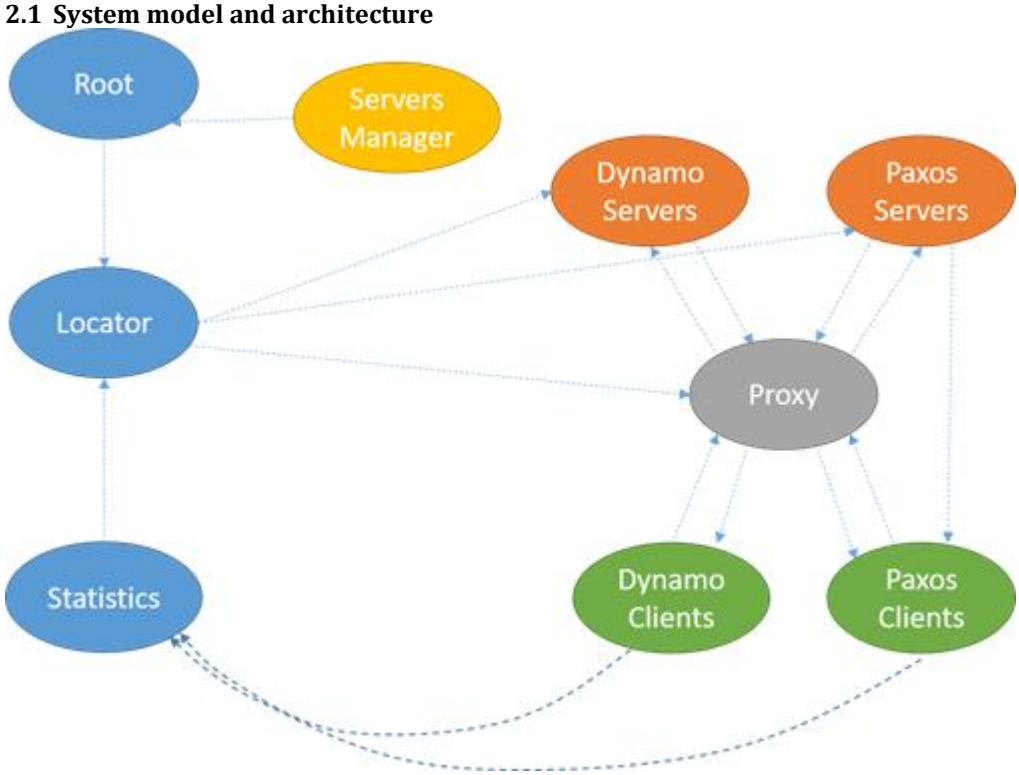

Figura 1 – Troca de mensagens entre componentes do sistema

Inicialmente é criado um actor Root que procede à leitura do ficheiro config.cfg.
É ainda criado um actor para servir de *Locator* a todos os outros e um actor *Statistics* para receber os dados estatísticos de cada cliente e guardá-los em ficheiros referentes a cada tipo de algoritmo usado (Multipaxos, Dynamo). Dados esses que serão posteriormente processados por um script em python.
O *Locator* recebe uma lista de servidores e fica à espera que o servidor de estatísticas se registe. Se receber pedidos de clientes antes desse registo proceder-se-á ao agendamento da resposta para quando o registo for feito.
Cada cliente criado enviará uma mensagem ao *Locator* com o pedido da lista de servidores e aquando da sua resposta irá começar a enviar operações ao servidor proxy que irá reenviar esse pedido para o(s) servidor(es) correcto(s). Assim, os clientes funcionam de forma muito semelhante para Multipaxos e para o Dynamo pois utilizam sempre o servidor proxy para redirecionar as operações.
O actor ServersManager irá proceder à alteração das variáveis do sistema, ou seja, alteração do número de servidores e do número de réplicas de cada chave.

2.2 Configurations and setup for operation

As configurações iniciais estão contidas no ficheiro config.cfg, a partir do qual decide o número de clientes que irão fazer pedidos para os servidores Multipaxos e para os servidores Dynamo, o número de instâncias possíveis, o número de servidores de cada tipo a serem criados, assim como o número de operações a serem escritas e lidas pelos clientes além da proporção de operações de cada tipo. Na configuração encontra-se também o número de servidores a serem adicionados e removidos assim como alterações ao número de réplicas de cada chave.

3. Repository Interface

A interface do repositório a ser usado pelo cliente, consiste nas seguintes operações:

Operação	Tipo	Argumentos	Descrição
insert	void	String Key, String elem	Insere o elemento "elem" no conjunto associado à chave "key", caso ainda não exista um conjunto associado à chave é criado ao inserir-se o primeiro elemento
remove	void	String Key, String elem	Remove o elemento "elem" do conjunto associado à chave "key", caso o conjunto fique sem elementos continuará a existir, apesar de estar vazio
isElement	boolean	String Key, String elem	Retorna true se o elemento "elem" estiver presente no conjunto associado à chave "key" e false caso contrário
list	List<String>	String Key	Devolve uma lista com os elementos presentes no conjunto associado à chave key, caso não exista elementos associado a essa key, retorna uma lista vazia

4. Strong consistency model using Multipaxos (1 page)

O Algoritmo de Paxos consiste num protocolo flexível e tolerante a falhas que pode ser usado para resolver problemas de consenso em sistemas distribuídos assíncronos. Dado um problema queremos garantir que:
-se todos os processos propõem v, então v é o único valor do output permitido (Validade),
-se dois processos estão correctos, então decidem o mesmo valor (Acordo)
-se o valor v for decidido, então v tinha de ser a proposta inicial de algum processo (Integridade)
Consideramos ainda que existem 3 tipo de processos, os proposers, acceptors e os learners, estes processos operam a uma velocidade arbitrária e assumimos que podem falhar por crash e depois recuperar, os processos comunicam entre si através de mensagens e estas são assíncronas podendo perder-se, serem duplicadas ou reordenarem-se mas nunca sendo corrompidas.
No entanto a propriedade de liveness (todos os processos correctos têm que decidir a certo ponto) não é garantida na presença de propostas concorrentes, para garantir esta propriedade implementou-se o multipaxos no qual se elege uma réplica como *Paxos leader* e se salta a fase de preparação pois o *leader* aceita logo o valor do cliente.

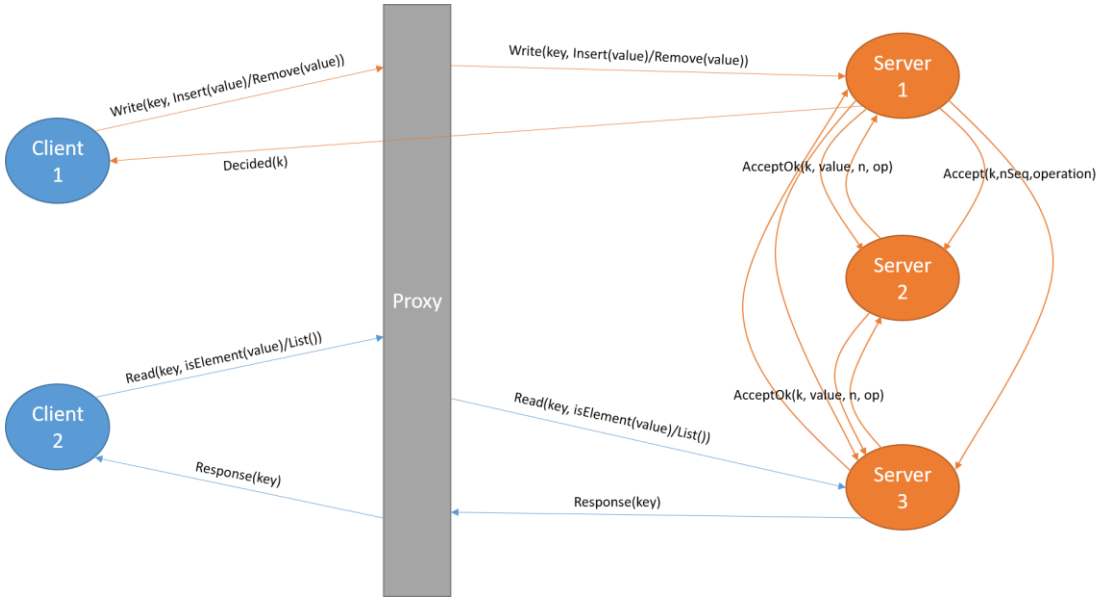

Figura 2 – Modelo baseado no Multipaxos

4.1 Multipaxos implementation

Partiu-se da implementação do Paxos entregue no TP1, na qual se alterou o que o servidor faz quando recebe um pedido de escrita, pois agora em vez de propor o valor aos outros servidores, verifica se é o *leader* daquela chave e em caso afirmativo aceita automaticamente o valor.
O cálculo do servidor *leader* é feito da mesma forma em todos os servidores e na proxy.

4.2. Multipaxos operation

O funcionamento da nossa implementação do Multipaxos consiste em permitir quatro operações feitas pelo cliente: insert(key,value), remove(key,value), isElement(key, value) e list(key).
Todas as operações feitas pelos clientes passam pelo servidor proxy, o qual redirecciona os pedidos para o *leader* correspondente à chave. No caso das operações insert e remove o resultado (Decided(k)) é devolvido directamente ao cliente, enquanto que nas restantes operações o resultado passa também pelo servidor proxy.

5. Eventual consistency and Dynamo (1 page)

A solução baseada no Dynamo, tem três tipos de intervenientes, clientes, servidores e um outro tipo de servidor, ao qual chamámos de proxy. Este proxy acaba por receber mensagens por parte dos clientes, com operações que estes pretendem que sejam executadas, enviando posteriormente para um certo quórum de servidores a operação devida, esperando obter resposta por parte desses servidores, para posteriormente poder responder de novo ao cliente que pediu a operação. A Figura 3 pretende ilustrar o funcionamento base.

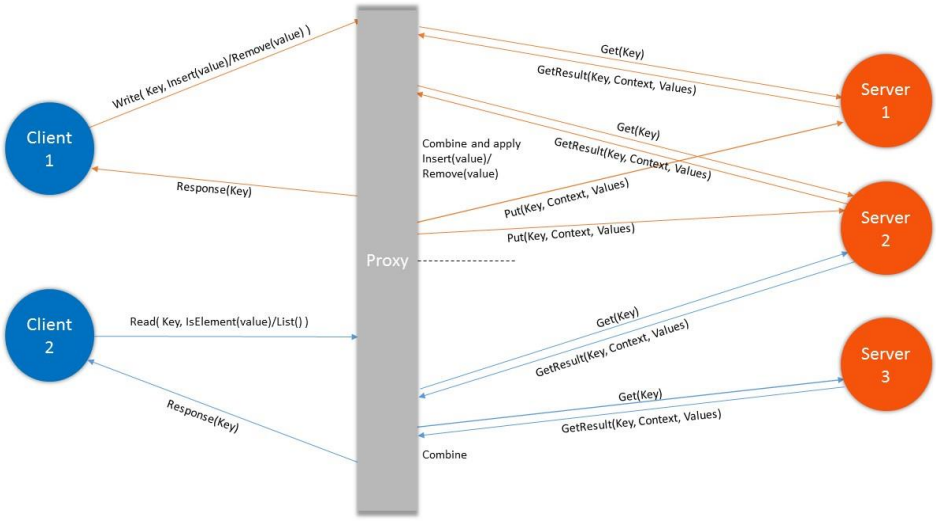

Figura 3 – Modelo baseado no Dynamo.

5.1 System operation

O funcionamento da nossa implementação do modelo baseado no Dynamo, consistiu em dar a possibilidade a um cliente de fazer quatro tipos de operações, insert(key,value), remove(key,value), isElement(key, value) e list(key). Estas operações são posteriormente enviadas para o servidor a que chamámos proxy, que age de forma bastante igual qualquer que seja a operação. No entanto no caso de ser uma operação de write, após combinar os valores e produzir um novo contexto(o nosso contexto consiste num modelo baseado em relógios vetoriais), envia uma mensagem do tipo Put(key,novo contexto, valores combinados) para o mesmo quórum que contactou inicialmente, ao fazer os Gets. Sendo que os servidores que recebem esta mensagem Put, alteram o contexto incrementando o campo do relógio vetorial que lhes corresponde. No caso do Read, após haver a combinação de valores, envia-se a resposta ao cliente, conforme a operação pedida.

5.2 Write/Read Quoruns

Para facilitar o funcionamento do sistema, decidimos que os quóruns de leitura e escrita seriam os mesmos para uma dada operação. Os servidores pertencentes ao quórum são selecionados aleatoriamente de um conjunto de servidores(optámos por selecionar uma maioria), que está associado a um hash code. Esse hash code pode ser produzido através da key a que a operação vai ser aplicada.

5.3 Use of consistent hashing

Usamos consistent hashing tanto no Proxy como nos restantes servidores. Em ambos os casos, foi implementado um método que recebe uma lista de servidores, e que constrói um Mapa<Int,Set[ActorRef]>, onde o inteiro é um hash code e o Set é um conjunto de K servidores, onde K é o número de vezes que uma chave deve ser replicada. Sempre que uma operação é feita, calcula-se o hash code da key a que a operação vai ser aplicada[6].

5.4 Management of results using CRDT sets

Os resultados que são obtidos por enviar mensagens Get(key) para os servidores, têm de ser combinados para garantir que se obtém um resultado final consistente. Para atingir este objetivo, implementou-se uma estrutura de dados, CRDT set(Add-wins set CRDT ou OR-Set), baseado na solução apresentada nas aulas[7], que combina todas as listas de valores obtidos dos servidores, pertencentes a um quórum.

6. Extensions

6.1 Dynamic membership management

A nossa solução para a gestão dinâmica de membros, passou pela implementação de um Actor, ServersManager, que executa todo o controlo sobre alterações que devem ser feitas ao ambiente de execução, em termos de servidores que entram e que saem, mas também relativamente ao número de vezes que uma chave deve ser replicada.

O nosso sistema inicialmente lê de um ficheiro de configuração, o qual para além de outros parâmetros, pode conter linhas que seguem a seguinte estrutura, "Dynamic Change:X,Y,Z". O X indica o número de servidores a serem adicionados, o Y representa o número de servidores a serem removidos e o Z representa o número de vezes que uma chave deve ser replicada(No enunciado do trabalho é referido como 'K').

O Actor ServersManager, a certo ponto recebe uma mensagem do Actor Locator, que lhe indica que pode começar a aplicar alterações ao ambiente do sistema. ServersManager, ao receber esta mensagem, começa a percorrer uma lista, que contém os parâmetros referentes às linhas de Dynamic Change do ficheiro de configuração. Por cada tuplo(nServersToAdd,nServersToRemove,kRep), presente na lista, aplica-se um método que envia mensagens ao Actor Root, a indicar quantos servidores deve adicionar e quais os servidores que deve remover. Escolheu-se que fosse o Root a fazer isto, visto que este é quem cria todos os outros Actors no sistema quando este é inicializado. Após a chegada da resposta por parte do Root, enviam-se mensagens para o Actor Locator, que passa depois a enviar mensagens tanto para o Proxy, como para os Servers, notificando-os da nova lista de servidores e do novo valor para K.

O nosso Sistema, utiliza um servidor entre o cliente e os outros servidores, como já foi mencionado anteriormente, e portanto, tanto no caso da solução Multipaxos como no caso da solução baseada no Dynamo, quando existe uma alteração dinâmica no número de servidores e no K, o Proxy e os servidores têm de efetuar operações.

6.2 Dynamic membership operation in the Multipaxos solution

É de notar, que embora tenha existido uma tentativa para resolver este problema da operação dinâmica de servidores para o Multipaxos, o resultado obtido da solução, não garante o correto funcionamento do programa em algumas ocasiões e portanto acabámos por deixar em comentário, dentro da classe ServersManager a linha que efetua esta operação. Esta linha pode ser descomentada, mas isso pode causar que os clientes Multipaxos não obtenham resposta a alguma operação e não avancem a sua execução. Obviamente isto não tem qualquer influência no funcionamento correto da solução Dynamo, que corre em simultâneo.

No caso do Multipaxos, o Proxy ao receber mensagem com a nova lista de servidores, calcula de novo o mapa que suporta o consistent hashing. Sempre que uma operação acaba e o Proxy pode responder ao cliente, essa operação é removida de um mapa, paxosProxyStates. Posteriormente é chamado um método, que percorre esse Mapa<Int,Tuple>, sendo que o inteiro é um id para a operação, que é criada quando um cliente envia uma operação para o Proxy, e o tuplo contém algumas informações, como o cliente que emitiu esta operação e o nome da operação. Ao percorrer esse mapa, verifica-se qual o tipo de operação(se é um insert/remove/isElement/list), e conforme o tipo, envia-se uma certa mensagem para o cliente a indicar que a operação acabou, removendo-se a entrada do paxosProxyStates, de forma a que o cliente possa continuar a execução. A mensagem enviada, por exemplo no caso de ter sido uma operação list(), é uma lista vazia, ou no caso do ter sido uma operação isElement() é false. Optámos por fazer desta maneira visto que mesmo que a resposta à operação original chegue ao

proxy, após ter havido a mudança no número de servidores, não é garantido que isso aconteça, e portanto quando o Proxy responde ao cliente, é como se a operação tivesse falhado.

No caso dos servidores, ao receberem a mensagem com a nova lista de servidores, recalculam o mapa que suporta o consistent hashing, passando depois a verificar em que entradas desse mapa, o próprio servidor se encontra no conjunto. Para esses casos em que o servidor se encontra dentro do conjunto, este percorre todos os outros servidores que também pertençam a esse conjunto, e envia-lhes uma mensagem GetStates(i), onde i é o índice no mapa.

Quando um servidor recebe uma mensagem desse tipo, este percorre os seus três mapas(proposerStates, acceptorStates, learnerStates) com keys, e calcula o hash code de cada key, sendo que, se o hash code de uma key for igual ao índice pedido na mensagem, adiciona-se para uma lista, essa entrada e no final responde-se ao servidor que lhe enviou a mensagem GetStates(i), com as três listas. Uma vez que o servidor inicial receba as mensagens com as listas de entradas, passa a verificar, quais as entradas que ainda não possui e adiciona-as aos seus mapas.

Existe ainda mais um passo que é executado, quando existe uma variação dos servidores ou do número de réplicas por chave. Percorre-se o mapa de learnerStates e para cada key, calcula-se o hash code, passando depois a consultar no mapa que suporta o consistent hashing. Se o próprio servidor já não pertencer ao conjunto remove-se essa key no mapa de learnerStates.

6.3 Dynamic membership operation in the Dynamo-based solution

A solução para o modelo baseado no Dynamo, também tem as duas vertentes do Proxy e dos servidores, sendo que apresenta um funcionamento bastante parecido à solução anterior. No entanto, quando ocorre uma mudança dos números de servidores, em vez de simplesmente responder logo aos clientes a indicar que a operação falhou, apagando-se os ids associados às operações que estavam pendentes, optou-se por recomeçar as operações, utilizando o mesmo id da operação. Neste caso, se os servidores conseguirem responder ao Get original, assim que as respostas chegarem ao Proxy, faz-se o processamento necessário e responde-se ao cliente, não existindo qualquer problema. Mesmo que agora se tenha repetido a operação, por haver possibilidade de esta não ter sido concluída, não existe problema, porque ao chegarem a segunda ronda de Get, não é feito nada no Proxy, visto que o id da operação já foi removido quando se respondeu ao cliente. Desta maneira garantimos que quando existe uma mudança dinâmica no ambiente do sistema, mesmo que tenhamos que repetir uma operação, causando algum atraso na resposta, o cliente irá sempre obter resposta às suas operações e não há risco de alterar o estado dos servidores duas vezes para uma mesma operação, visto que o id da operação é removido assim que se responde ao cliente.

No lado dos servidores, assim que recebam a mensagem com a nova lista de servidores, recalculam o mapa que suporta o consistent hashing, e procedem da mesma maneira que a solução do Multipaxos, mas neste caso só se efetuam alterações num mapa, visto que só é utilizado um mapa com os estados, para a solução baseada no Dynamo.

7. System model and implementation (~1-2 pages)

Para a implementação deste projecto utilizamos o software Eclipse com bibliotecas externas para poder usar o ambiente de programação “akka” e como linguagem o “Scala”.

O modelo de programação em akka [8] é baseado em concorrência de actores, o que nos permite usar linguagens concorrentes e funcionais para desenvolver sistemas distribuídos, resistentes a falhas, com grande disponibilidade, "soft real time", com a capacidade de retirar e substituir componentes desse sistema.

Caracterizado por ter um elevado nível de abstração em que só temos de pensar no flow de mensagens entre actores, podendo assim dedicar-nos à detecção e recuperação de erros. Como benefícios do uso do akka temos ainda, a optimizada utilização do CPU, baixa latência, alta largura de banda e possibilidade de escalabilidade. Tudo isto simplifica o desenvolvimento de aplicações concorrentes escaláveis e distribuídas. A versão usada do akka foi a 2.4.14 .

Como linguagem de programação usámos o Scala em vez de Java, apesar de o Scala ser considerada uma linguagem mais difícil, a curva de aprendizagem vale o investimento. O Scala tem uma escrita mais complexa mas acaba por ser mais proveitoso escreve um linha mais complexa em Scala do que 20 linhas simples em Java, que tem estrutura de escrita mais simples. A versão do Scala usado é a 2.11.

Para facilitar a disponibilidade do nosso código entre o grupo e facilitar o trabalho paralelo da equipa, recorremos ao uso de um repositório GitHub, usando programas como o sourceTree ou extensões no Eclipse para trabalhar sobre o mesmo.

Para tratar e representar os dados obtidos usou-se a linguagem Python 2.7, que através de bibliotecas como o Numpy permite a manipulação fácil de grandes quantidades de dados.

8. Experimental evaluation (1 page)

Para efetuar os testes, o programa foi executado várias vezes numa máquina em repouso com a configuração pedida no enunciado: 10000 operações sobre 100 chaves diferentes com a proporção: insert – 40%, remove – 10%, isElement – 40%, list – 10%. Executadas por três clientes de cada tipo (Multipaxos e Dynamo) sobre o mesmo sistema.
Pretende-se capturar os tempos de cada operação observados pelos clientes, calculando posteriormente o número de operações efectuadas por segundo e permitindo comparar as performances de cada algoritmo implementado.
Em cada operação efetuada por um cliente, este calcula a diferença entre o tempo em que enviou o pedido e recebeu a resposta, guardando o resultado.
Para poder avaliar os testes feitos implementámos um ator *Statistics*, que recebe mensagens com um conjunto de valores provenientes dos clientes, além da informação sobre o algoritmo usado nos pedidos desse cliente. Esse ator organiza esses dados em listas e guarda-as em quatro ficheiros dedicados aos dados de operações de write e de read em ambos os algoritmos.
Posteriormente esses ficheiros são alimentados como entrada de um script python que calcula os diagramas de caixa de cada ficheiro.

Gráfico 1 – Comparação dos tempos de escrita

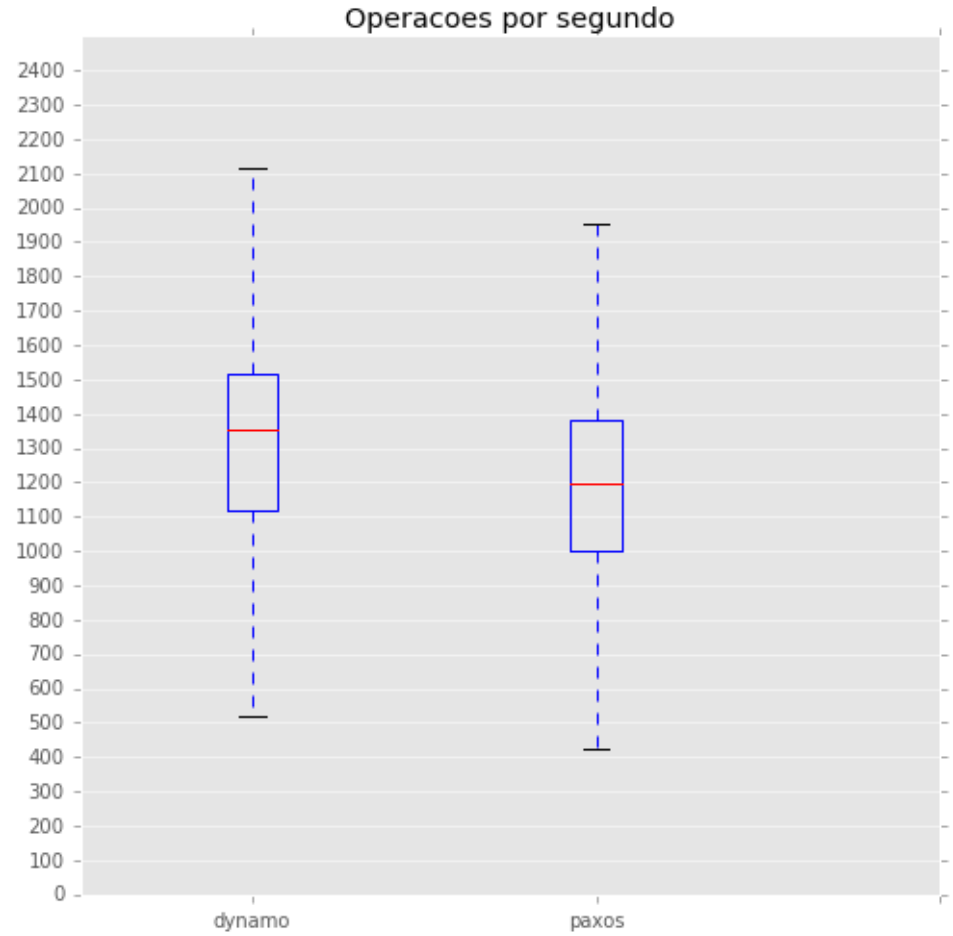

Gráfico 2 – Comparação dos tempos de leitura

Como podemos observar nos gráficos 1 e 2, o algoritmo dynamo mostrou-se mais eficiente em todos os testes efectuados embora estivéssemos à espera de uma diferença maior.
Repetindo os testes com diferentes valores de servidores e clientes obtêm-se resultados semelhantes assim como variando o número de operações efetuadas, embora com poucos clientes essa diferença tenha tendência a aumentar. Pelo que pensamos que uma das possíveis causas para estes resultados seja a centralização de todos os pedidos a passar no servidor proxy.
Este resultado vai de encontro com o que estudámos na cadeira pois o dynamo envolve menos trocas de mensagens que o paxos, mesmo implementando a variante multipaxos este continua a ter muito mais mensagens que o dynamo.

9. Conclusions

Tendo em conta o projecto realizado podemos concluir que este trabalho foi uma grande mais valia, em extensão de tudo o realizamos no primeiro trabalho prático, pudemos consolidar e desenvolver as ideias dos conhecimentos teóricos obtidos anteriormente, em relação ao algoritmo de Paxos/Multi-Paxos) e Dynamo.

A progressiva elaboração do trabalho prático permitiu que conseguíssemos consolidar bastante os nossos conhecimentos na área dos sistemas distribuídos, tanto a nível mais geral como a alguns níveis mais específicos. O facto de termos de lidar com as situações de falhas, torna o problema mais complexo, mas é extremamente relevante por ser uma aproximação da realidade. Por fim, a parte de análises do comportamentos dos

algoritmos também se verificou muito importante para construirmos uma ideia mais sólida dos mesmos.

Ficámos bastante satisfeitos com o resultado final de todo o trabalho investido na cadeira, pois temos a completa noção do quanto importante estes conhecimentos se podem revelar na nossa futura área de trabalho.

Pensamos ter atingido todos os objetivos propostos inicialmente para a realização deste trabalho, mesmo tendo um ligeiro problema nas variações do Multi-Paxos, que pode causar inconsistência no desempenho do mesmo.

References

[1] Distributed Systems and Algorithms 2015/2016 - MIEI Course, DI-FCT-UNL, TP2 ASD-Project2-v1, Nov 2016

[2] Distributed Systems and Algorithms 2015/2016 - MIEI Course, DI-FCT-UNL, Material from lecture classes, Oct 2016

[3] Leslie Lamport, *Paxos Made Simple*, Technical Report, Nov 2001 (available in http://research.microsoft.com/en-us/um/people/lamport/pubs/paxos-simple.pdf)

[4] Distributed Systems and Algorithms 2016/2017 - MIEI Course, DI-FCT-UNL, Lecture 7, Eventual Consistency, Dynamo, CRDT, Nov 2016

[5] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshal, P., Vogels, W. (Oct 2007). Dynamo: amazon's highly available key-value store, 205-220. SOSP 2007.

[6] Distributed Systems and Algorithms 2016/2017 - MIEI Course, DI-FCT-UNL, Lecture 9, Teorema CAP, Escalabilidade no Dynamo, Sistemas P2P, Nov 2016

[7] Distributed Systems and Algorithms 2016/2017 - MIEI Course, DI-FCT-UNL, Lecture 8, Consistência Causal, CRDTs, Teorema Cap, 42-45, Nov 2016

[8] Distributed Systems and Algorithms 2014/2015 - MIEI Course, DI-FCT-UNL, LAB 1 Setting the Scene: Programming with akka