FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Key value store using State Machine Replication and Paxos

Abstract

Este projecto tem como objetivo o desenvolvimento de uma *key-value store* com consistência forte. Para isto foi utilizada uma abordagem de *state machine replication* que recorre ao algoritmo Paxos. Os resultados obtidos demonstram a *performance* da solução, tendo-se testado até 100 clientes a executar operações sobre 10 réplicas. Não se tendo atingido a *wall* caraterística dos gráficos latência/*throughput*.

1. Introdução

Algoritmos de *state machine replication* (SMR) garantem propriedades benéficas para um sistema de replicação distribuído, destacando-se a ordenação total das operações e tolerância a falhas. Estas propriedades são asseguradas através do uso de *quorums* de maioria, enquanto fornecem ao cliente a ilusão que o serviço não se encontra replicado.

Dado o contexto anterior, este relatório apresenta uma base de dados chave valor que recorre a SMR e ao algoritmo Paxos para garantir consistência forte.

O presente relatório segue a seguinte estrutura: a Secção 2 descreve, brevemente, a solução implementada; o pseudo-código e os argumentos de correção das várias camadas existentes em cada nó da rede é realizado na Secção 3; a avaliação experimental é efetuada na Secção 4; na Secção 5 apresenta a conclusão deste relatório; finalmente, como anexo estão presentes os dados que foram a base da representação gráfica.

2. Overview

A solução obtida para solucionar este problema é composta por duas entidades: cliente e réplica. Cada uma destas apresenta duas camadas.

A camada de topo de cada réplica é SMR que fornece os seguintes pedidos:

I. weakGet(chave): return <valor>;
II. strongGet(chave): return <valor>;
III. put(chave, valor): return <valor antigo>;
IV. addReplica(replica): return <indíce>;
V. removeReplica(replica): return <índice>.

As duas primeiras operações permitem consultar o estado da base de dados chave valor, sendo que a única distinção encontra-se nas garantias de causalidade oferecidas por cada uma. A primeira, weakGet, garante consistência fraca e por isso não é necessário ser ordenada. A segunda, strongGet, garante consistência forte impossibilitando o cliente de observar estados divergentes. Já a operação de put permite modificar o estado da *store*. As últimas duas operações permitem a gestão das réplicas da SMR. Para se obter *consensus*

André Lopes - 45617
Nelson Coquenim - 45694

sobre qual a operação a executar em todas as réplicas da SMR existe uma camada que contém o algoritmo Paxos, sendo que este disponibiliza os pedidos *Propose*(valor) e *Decided*(valor).

O cliente dispõe duma camada que contém a sua lógica aplicacional denominada na Figura 1 como Test App sendo que esta realiza operações na SMR através da camada Key Value Client que atua como um intermediário.

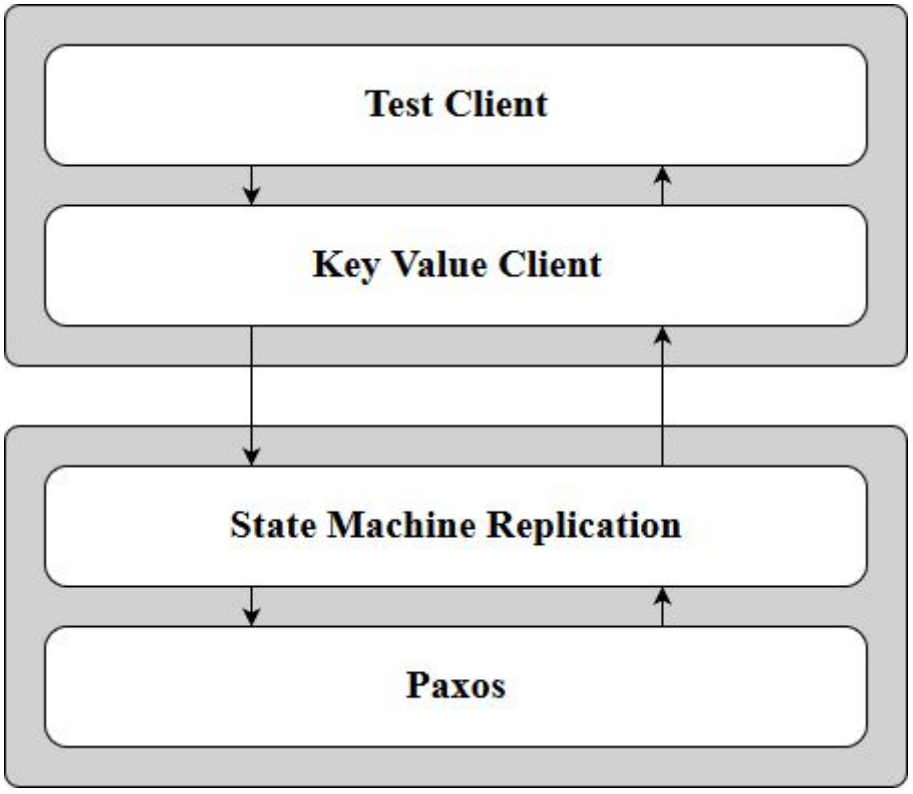

Figura 1: Representação das várias camadas existentes num nó da state machine (rectângulo de baixo) e no cliente (rectângulo de cima).

3. Camadas

3.1. State Machine Replication

A SMR tem que garantir que as operações efetuadas na base de dados e as operações de gestão de réplicas são efetuadas pela mesma ordem em todas as réplicas. Para tal vai ser necessário a existência de uma estrutura de dados que guarde o histórico de operações que já foram vistas. Assim sendo, após a receção de um pedido de um cliente, a réplica propõe a operação através do paxos para um lugar específico do histórico. Note-se que é necessário usar uma instância diferente do paxos para cada posição do histórico, dado que, o paxos só decide um único valor.

Quando uma réplica recebe uma decisão do paxos existem duas possibilidades. A decisão pode ser a operação que este propôs, ou então outra operação proposta por outra

réplica. Na primeira hipótese é possível proceder-se à execução da operação enquanto que na segunda, visto que foi rejeitada a proposta da réplica, é necessário propor outra vez para a próxima posição do histórico disponível.

Um aspecto importante relativamente à execução de uma operação é a necessidade de conhecer-se o histórico de operações anteriores, caso contrário, a execução da mesma é impossível.

Adicionalmente, como uma réplica pode receber várias operações para executar, é necessário guardar as operações numa fila para que não se perca os pedidos dos clientes.

Outra consideração importante subsiste nas operações de gestão das réplicas. Na adição de uma nova réplica à *membership* da SMR é necessário enviar a esta o histórico de operações senão a nova réplica não consegue recriar a *store* nem a *membership* atual da SMR.

Finalmente, tanto na adição como na remoção de réplicas é necessário atualizar a camada do paxos para não se invalidar o *quorum* de maioria deste e por conseguinte tomar a decisão errada.

Pseudocódigo da State Machine Replication

```
Interface:
    Requests:
        WeakGet (mid, key)
        StrongGet (mid, key)
        Put (mid, key, value)
        AddReplica (mid, r)
        RemoveReplica (mid, r)
    Indications:
        Reply(mid, returnValue)
State:
    history          // array that contains the history of operations
                     // [i, {(op, returnValue, mid, executedclient, replica),...}]
    current          // last index used in history
    store            // map[key, value]
    toBeProposed     // queue with ops to be proposed (op, mid, client)
    proposed         // set of mids of proposed operations
    replicas
    historyProcessed //prevents the execution of history more than one time
    lastExecuted     // last executed operation index in history

    Upon Init(_replicas_) do:
        history ← {}
        current ← 0
        toBeProposed ← {}
        proposed ← {}
        historyProcessed ← false
```

```
        replicas ← _replicas_

    Upon Receive(HISTORY, history, index) do:
        Call ExecuteHistory(_history_, index)

    Upon Receive(WEAK_GET, s, key, mid) do:
        Trigger Send(GET_REPLY , s, map[key], mid)

    Upon Receive(OP = PUT | STRONG_GET | ADD_REPLICA | REMOVE_REPLICA
      ,s, mid) do:
        If mid ∉ proposed then
            proposed ← proposed U {mid}
            toBeProposed ← toBeProposed U {(OP(args), mid, client)}

            If #toBeProposed = 1 then //not waiting for any propose
                current ← FindValidIndex()
                Trigger Propose(toBeProposed , current)
        Else
            For each batch ∈ history
                For each (op, returnValue, mid, executed, client, replica) ∈ batch
                    If mid = _mid_ ∧ executed = true then
                        Trigger Send(REPLY, s, op, mid, returnValue)
                        break

    Upon DecisionDelivery(DecisionDelivery, ops, i) do:
        history[i] ← ops
        If #toBeProposed = 0 ∧ ops ⊆ toBeProposed then  //2nd cond: proposed by him
            toBeProposed ← toBeProposed \ ops
        If #toBeProposed != 0 then
            current ← FindValidIndex()
            Trigger Propose(toBeProposed, current)
        If PreviousCompleted(i) then
            For each op ∈ ops
                Call ExecuteOp(op, i)

    Procedure ExecuteOp(op, i) do:
        toReturn ← {}
        If op = PUT then
            toReturn  ← store[key]
            store[key] ← value
        If op = STRONG_GET then
            toReturn  ← store[key]
        If executableOp = ADD_REPLICA
```

```
                    toReturn ← i
                    replicas ← replicas U {r}
                    Trigger Send(UPDATE_REPLICAS, paxosProposer, replicas)
                    Trigger Send(HISTORY, r, history, i)
            If executableOp = REMOVE_REPLICA
                    toReturn ← i
                    replicas ← replicas \ {r}
                    Trigger Send(UPDATE_REPLICAS, paxosProposer, replicas)
            op.executed ← true
            op.returnValue ← oldValue
            history[i] ← history[i] U op
            If op.replica = myself V op.replica ⊊ replicas      // replica of client failed
                    Trigger Send(REPLY, op.client, op)     //op includes returnValue and mid

Procedure ExecuteHistory(_history_, index) do:
        If historyProcessed = false then
            For Each ops ∈ _history_ do:
                For Each op ∈ ops do:
                toReturn ← {}
                If op = PUT then
                        toReturn ← store[key]
                        store[key] ← value
                If op = STRONG_GET then
                        toReturn ← store[key]
                If executableOp = ADD_REPLICA
                        toReturn ← i
                        replicas ← replicas U {r}
                        Trigger Send(UPDATE_REPLICAS, paxosProposer, replicas)
                If executableOp = REMOVE_REPLICA
                        toReturn ← i
                        replicas ← replicas \ {r}
                        Trigger Send(UPDATE_REPLICAS, paxosProposer, replicas)
            history[i] ← history[i] U ops
        historyProcessed ← true

Procedure PreviousCompleted(index) do:
        tempLastExecuted ← 0
        For each (i, ops) ∈ history ∧ lastExecuted < i < index do:
                If history[i] != ⊥ then
                        completed ← false
                If history[i] != executed then
                        For each op ∈ history[i] do:
                                ExecuteOp(op, i)
```

```
                    tempLastExecuted ← i
        lastExecuted ← tempLastExecuted
        return completed

Procedure FindValidIndex() do:
        tempIndex ← 0
        // lastIndex is the highest filled slot in history
        For each slot ∈ history ∧ current < i <= lastIndex do: history
            If history[i] = ⊥ then
                    return i
        return lastIndex + 1
```

3.1.1. Argumentos de correção

Pela a construcção do código e incluindo as garantias dados pelo o Paxos pode-se afirmar que todas as réplicas executam as operações pela mesma ordem, convergindo para o mesmo estado.

3.2. Paxos

O paxos permite resolver o problema do *consensus* em sistemas distribuídos assíncronos nos quais poderá haver falhas de pelo menos um processo. Este relaxa a propriedade de *termination* do *consensus* para garantir que num momento em que o sistema se comporta de forma síncrona é possível decidir-se um valor proposto.

Esta camada permite que a SMR submeta e receba operações para sem executadas segundo a ordem determinada pelo paxos.

Importa salientar que, para lidar com o problema de se estar a criar múltiplos atores Paxos no *scala* para fazer várias propostas alterou-se o pseudo código do paxos para que este suportasse várias instâncias, a alteração óbvia foi colocar todas as variáveis indexadas pelo número da instância, garantindo assim a separação entre várias instâncias, mas evitando a gestão de múltiplos atores no *scala*.

Pseudocódigo do Paxos

PROPOSER

```
Interface:
        Requests:
                    Propose (value, i)
        Indications:
                    DecisionDelivery(value, i)

State
        replicas
        sn[i]           // sequence number of the proposed value
```

```
        value[i]          // value to be proposed
        prepares[i]       // number of prepareok received
        accepts[i]        // number of acceptok received
        highestSna[i]     // highest sna seen so far in a prepareok msg
        lockedValue[i]    // value corresponding the highestSna
        prevMajority      // true if already exists a decision

Upon Init(_replicas_) do:
        replicas← _replicas_
        sn[i] ← {}
        value[i] ← {}
        prepares[i] ← {}
        accepts[i] ← {}
        highestSna[i] ← {}
        lockedValue[i] ← {}
Upon Receive(UPDATE_REPLICAS, _replicas_) do:
        replicas← _replicas_

Upon Propose(v, i) do:
        Call ResetState()        //because ResendOp timer also uses this
        value[i] ← v
        sn[i] ← getSN            // chooses unique sn larger than the ones seen so far
       For Each a ∈ acceptors  do:
                Trigger Send(PREPARE, a, sn, i)
        Setup Periodic Timer PrepareTimer(PREPARE_TIMEOUT, i)

Procedure ResetState() do:
        prevMajority ← false
        prepares ←0
        accepts ←0

Upon Receive(PREPARE_OK, s, sna, va, i, snSent) do:
        If snSent = sn[i] then     ///in order to ignore past prepares
                prepares[i] ← prepares[i] + 1
                if sna > highestSna[i] ∧ va != ⊥ then
                        highestSna[i] ← sna
                        lockedValue[i] ← va
                if prepares[i] > #π/2 && !prevMajority[i] then
                        prevMajority = true
                        Cancel PrepareTimer[i]
                        if(lockedValue[i] != ⊥) then
                                value ←  lockedValue[i]
                        For each a ∈ acceptors  do:
                                Trigger Send(ACCEPT, a, sn, value, i)
                        Setup Periodic Timer AcceptTimer(ACCEPT_TIMEOUT, i)
```

Upon Receive(ACCEPT_OK, s, sn, i, snSent**) do:**
    If snSent = sn[i] **then** //in order to ignore past accepts
        accepts[i] ← accepts[i] + 1
        if #accepts[i] > #π/2 **then**
            Trigger Send(LOCKED_VALUE, myLearner, value, i)
            Cancel AcceptTimer[i]

Upon PrepareTimer(t, i**) or AcceptTimer(**t, i**) do:**
    Trigger Propose(value, i**)**

ACCEPTOR

State:
    np[i] // highest prepare
    na[i] //highest accept
    va[i] //highest accept val

    Upon Init() do:
        np[i] ← ⊥
        na[i] ← ⊥
        va[i] ← ⊥

    Upon Receive(PREPARE, s, n, i**) do:**
        If n > np[i] **then**
            np[i] ← n
            Trigger Send(PREPARE_OK, s, na[i], va[i], n)

    Upon Receive(ACCEPT, s, n, v, i**) do:**
        If n >= np[i] **then**
            na[i] ← n
            va[i] ← v
            Trigger Send(ACCEPT_OK, s, n, i)

Learner

State:
    decided[i] // flag that determines if there was already a decided value or not

    Upon Init(_replicas_**) do:**
        decided ← {}

    Upon Receive(LOCKED_VALUE, s, v, i**) do:**
        If decided[i] = false **then**
            decided[i] ← true

Trigger DecisionDelivery(v, i)

3.2.1. Argumentos de correção

Segundo as garantias do Paxos, o pseudo código apresentado para o mesmo respeita todas as seguintes propriedades: i) C2: *Validity*: Se um processo decide v, então v foi proposto por algum processo. De facto, se observarmos o pseudocódigo a única maneira de uma valor ser decidido é se primeiramente for proposto. Deste modo, pela construção do algoritmo verifica-se esta propriedade; ii) C3 *Integrity*: Nenhum processo decide duas vezes. Apenas é possível decidir uma vez, visto que existe um booleano no pseudocódigo do *learner* que impossibilita múltiplas decisões; iii) C4 *Agreement*: Nenhum par de processos correctos decide valores distintos. Esta propriedade é impossível de quebrar devido à construção do algoritmo, em especial das mensagens *Prepare Ok* que retornam o valor que o respetivo *acceptor* já aceitou. Sendo assim o *proposer* que receber um *Prepare Ok* com um valor diferente de *bottom* é obrigado a trocar o seu valor pelo valor já aceite. Eliminando assim a possibilidade de se vir a decidir valores distintos.

De salientar que o Paxos não garante *liveness* havendo a possibilidade de, no decorrer da primeira fase de *prepares,* múltiplos *proposers* originarem um bloqueio devido à sucessiva sobreposição de *prepares* com números de sequência cada um maior que o anterior.

3.3. Key Value Client (subtítulo)

Esta camada disponibiliza uma interface ao cliente final que permite a manipulação da SMR, incluindo a adição e remoção de réplica bem como a manipulação do estado da *store*.

Adicionalmente esta garante a propriedade de *safety* de ser impossível receber a mesma operação mais do que uma vez através de uma estrutura de dados que contém os identificadores únicos das mensagens já recebidas. Para além disso em situações em que se não obtém o resultado/confirmação da operação submetida passado um determinado tempo esta camada é responsável por reenviar o pedido para outra réplica (escolhida de forma aleatória).

Pseudocódigo do Key Value Client

Interface:
    Requests:
        StrongGet (key**)**
        WeakGet(key**)**
        Put (key, value**)**
    Indications:
        Delivery(op, returnValue**)**

State:
    replicas

```
delivered

Upon Init(_replicas_) do:
    replicas = _replicas_

Upon Get(key) do:
    If mid ∉ delivered then
        mid ← generateUID(ip, port, timestamp)
        r ← chooseReplica(replicas)
        Trigger Send(WEAK_GET, mid, key)

Upon Receive(STRONG_OP = PUT | STRONG_GET | ADD_REPLICA |
    REMOVE_REPLICA, args) do:
    mid ← generateUID(ip, port, timestamp)
    r ← chooseReplica(replicas)
    Trigger Send(STRONG_OP, mid, args)
    Setup Periodic Timer ResendOp(T, STRONG_OP, args)

Upon ResendOp(OP, mid, args) do:
    If mid ∉ delivered then
        Trigger Send(OP, mid, args)
    Else
        Cancel Timer ResendOp(mid)

Upon Receive(REPLY, mid, returnValue) do:
    If mid ∉ delivered then
        delivered ← delivered U {mid}
        Trigger Delivery(returnValue)
```

3.3.1 Argumentos de correção

Esta solução apresenta-se como correcta devido à construção da mesma e porque não quebra as propriedades que as camadas anteriormente descritas fornecem. Adicionalmente, verifica-se a inexistência de repetição de operações graças à estrutura de dados *delivered* que armazena os identificadores únicos de operações que já foram executadas.

3.4. Test App

Por último, esta camada irá conter a lógica deseja pelo o cliente. Sendo que no desenvolvimento deste sistema foi desenvolvido uma aplicação de teste para avaliar a correcção da solução bem como o seu desempenho. A descrição dos testes realizados encontra-se na Secção 4.

4. Avaliação Experimental

De forma a avaliar experimentalmente a solução foi desenvolvida uma aplicação teste que executa operações sobre o sistema replicado. Os objetivos desta avaliação passam por analisar experimentalmente a correção da solução e por outro lado avaliar a sua *performance* através de testes em que se varia a carga sobre o sistema. É importante também referir que todos os processos correm em JVMs distintas.

No que toca a avaliar a correção, a lógica do teste passa por executar diversos clientes a executar escritas de forma sequencial sobre a mesma chave. Para além disto, cada escrita por parte de cada cliente é executada numa réplica aleatória de forma a aumentar a probabilidade existirem erros de sincronização. Após todos os clientes pararem de executar operações, ir-se-á obter a *store* de cada réplica e o histórico de operações, avaliando se todas as réplicas têm um estado igual.

Em relação à avaliação da *performance*, tal como a da correção, foi feita com diversos clientes a executar operações de forma sequencial sobre réplicas aleatórias. Desta feita os testes foram tanto de leituras com consistência fraca como de escritas. De forma a obter as métricas necessárias, cada cliente guarda o número de operações executadas e a latência obtida em cada uma delas. No fim da execução, calcula-se para cada cliente a média das latências e a quantidade de operações por segundo (*throughput*). Tendo estes dados para cada um dos clientes é possível avaliar a *performance* do sistema desenvolvido, fazendo a média das latências e somando os *throughputs*. Com esta informação consegue-se analisar a escalabilidade do sistema, o impacto do aumento do número de réplicas e as diferenças de *performance* de uma solução com e sem *batching*.

4.1 Condições Experimentais

Os testes executaram durante 20 segundos, sendo que ambos foram avaliados variando tanto o número de clientes como o do número de réplicas. Num dos testes fixou-se o número de clientes em 1 e variou-se o número de réplicas de forma a analisar o impacto da sincronização entre as mesmas. No contexto dos testes de carga, fixou-se o número de réplicas em 10, variando-se o número de clientes entre 1 e 100 com incrementos de 10. Por fim, analisou-se o impacto do *batching* na solução com a configuração referida na frase anterior.

No que diz respeito à máquina utilizada, recorreu-se a um fornecedor de serviços de *cloud*, neste caso o Azure. Esta máquina tem 32 vCPUs e 448GB RAM, o que permitiu avaliar a solução num contexto que permita avaliar verdadeiramente a escalabilidade. De referir que esta é uma forma de emular a distribuição dos nós em diferentes máquinas já que cada um destes é uma JVM.

4.2 Discussão de Resultados

Após a extração e tratamento dos resultados obtidos nos testes descritos anteriormente, foi possível inferir algumas conclusões.

Primeiramente, foi possível observar pela realização dos vários testes, a correção da solução. Uma vez que, após o término destes, o estado (histórico de operações, *store* e *membership*) das várias réplicas era idêntico.

Os resultados obtidos no teste à *performance* da solução tendo em conta a variação do número de réplica consta no Gráfico 1. Pode-se observar que o ponto com um menor desempenho é o primeiro ponto, mais à esquerda, com cerca de 40 milissegundos de latência e 25 operações por segundo. Este ponto corresponde ao uso de 10 réplicas sendo os seguintes, decrementos até 3 réplicas. Este gráfico revela que o aumento do número de réplicas contribui negativamente para o desempenho da solução devido ao custo da sincronização do estado.

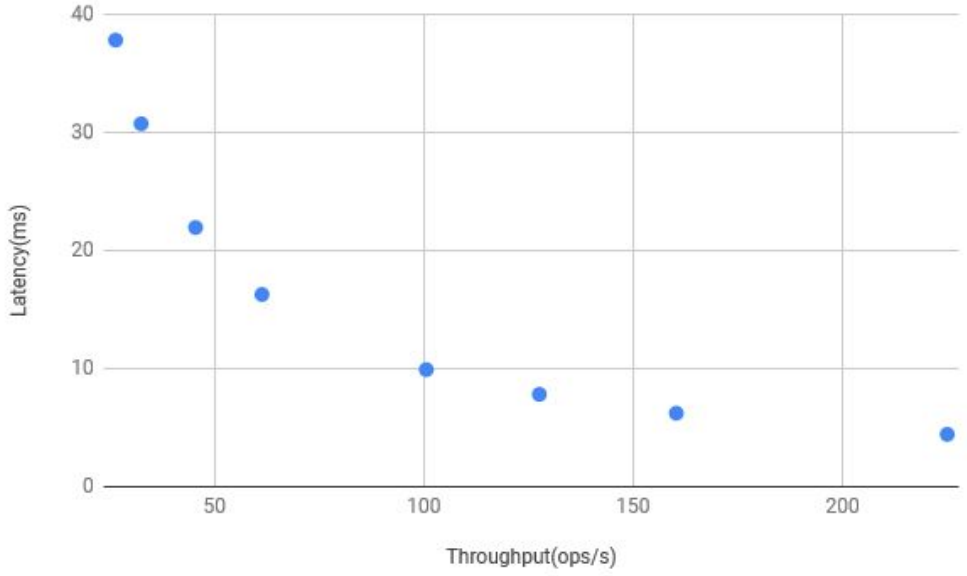

Gráfico 1: Latência e throughput variando o número de réplicas utilizadas entre dez (primeiro ponto à esquerda) e três (primeiro ponto à direita). Vinte segundos de puts consecutivos.

No Gráfico 2 pode se observar o desempenho da solução, medido em operações por segundo, com o aumento do número de clientes. Conclui-se que não existe um aumento proporcionalmente directo com o número de clientes, uma vez que duplicando o número de clientes não duplica o *throughput*. Todavia é observável que o declive é cada vez maior a partir dos 70 clientes, o que significa que ainda não se está a aproximar da *wall*, pois aí os declives irão decrescer rapidamente, aproximando-se de uma reta horizontal. Portanto ainda é possível adicionar clientes e obter *throughputs* cada vez maiores.

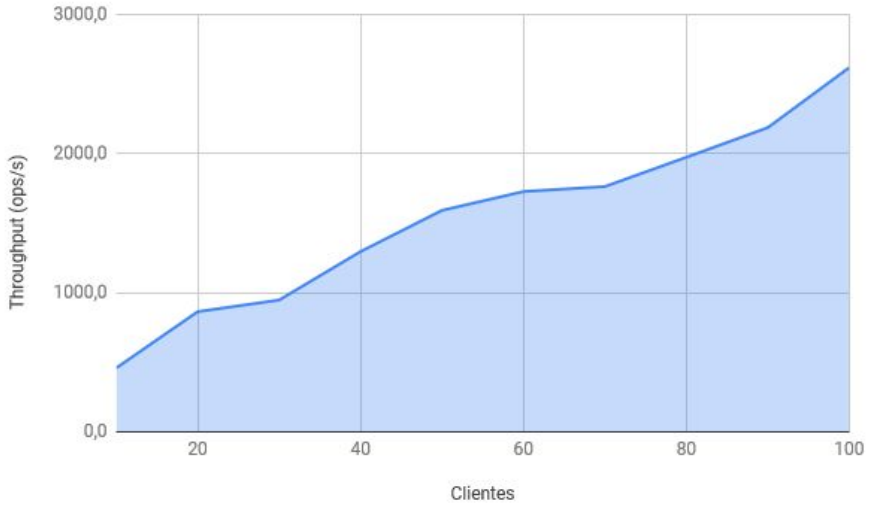

Gráfico 2: Throughput, em operações por segundo, variando o número de clientes entre dez e cem em incrementos de dez clientes por teste. Vinte segundos de puts com *batching* de operações.

Foi também possível medir o impacto de recorrer ao Paxos para se decidir sobre a ordem de uma ordenação *versus* operações que não necessitam de ser ordenadas. Sendo assim, o Gráfico 3 compara a *performance* entre a operação put, que é oferecida sobre um modelo de consistência forte e a operação *weak* get onde garante-se consistência fraca.

Existe uma clara diferença no *throughput* e na latência entre ambas as operações o que comprova que garantir consistência forte oferece grandes vantagens tais como, tornar mais fácil raciocinar sobre o estado do sistema e a possibilidade de adicionar mais garantias sobre o mesmo. No entanto, a eficiência não é óptima porque requer consultar um *quorum* de réplicas para executar uma operação e não é possível garantir *liveness*.

Finalmente, no Gráfico 4, pode observar-se a latência e o *throughput* com e sem *batching* de operações, durante vinte segundos, onde são efetuados apenas puts. Em ambas as linhas, o primeiro ponto à esquerda é com 10 clientes e os restantes são incrementos de 10 clientes, ou seja, o último ponto representa os resultados com 100 clientes. Observa-se que apesar do aumento no número de operações efectuadas, o tempo de resposta a uma operação vai ficando cada vez maior. Esta observação deve-se a que, existindo mais clientes, cada um terá que esperar mais pela execução da sua operação, pois para o mesmo número de réplicas aumenta-se o processamento que cada uma terá que fazer.

Contudo, não se chega a uma fase em que o aumento de clientes não leva a um aumento de *throughput* (*wall*). De assinalar as diferenças consideráveis entre usar uma estratégia de *batching,* pois o aumento do *throughput* é acompanhado de um menor aumento de latência do que sem esta técnica.

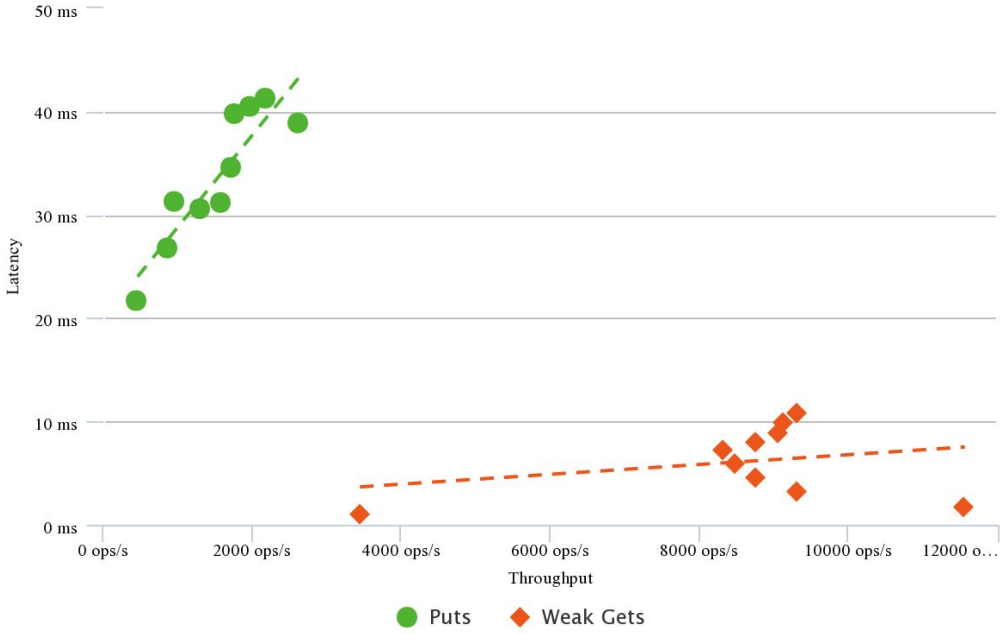

Gráfico 3: Latência e throughput variando o número de clientes entre dez e cem em incrementos de dez clientes por teste. Cada teste demora vinte segundos, onde são realizados puts recorrendo a uma estratégia de *batching* no caso da recta verde, e *weak* gets na recta laranja.

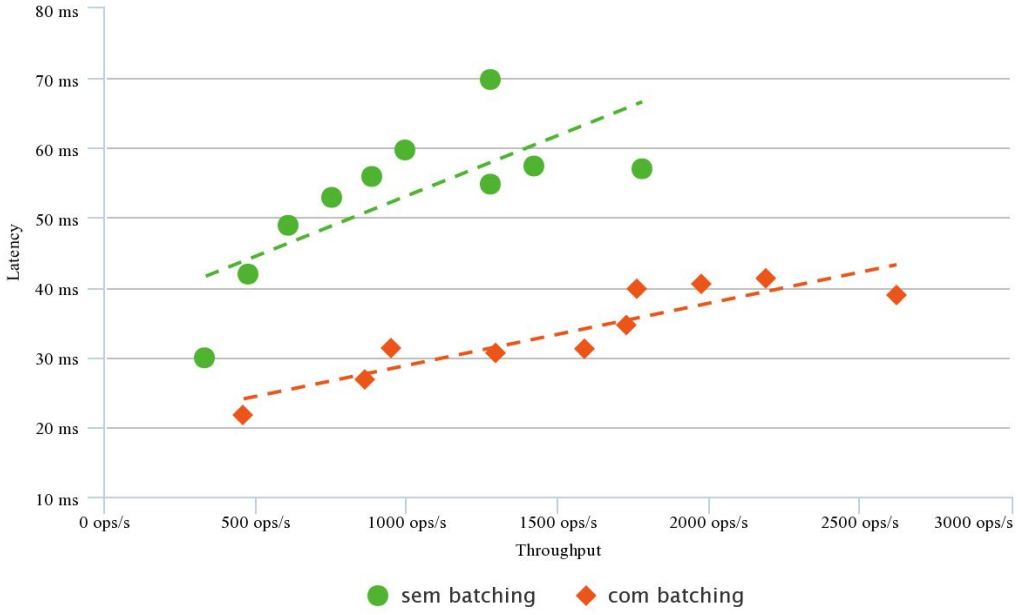

Gráfico 4: Latência e throughput variando o número de clientes entre dez e cem em incrementos de dez clientes por teste. Vinte segundos de puts sem batching *versus* com *batching* de operações.

Conclusão

No final do presente relatório importa retirar conclusões. Conseguiu-se elaborar uma solução correcta de state machine replication que aguenta a carga de pelo menos 100 clientes com 10 réplicas. Contudo não se descobriu o valor máximo de *throughput* que este sistema suportaria.

Verificou-se que a utilização de algoritmos que oferecem um modelo de consistência forte em sistemas de replicação distribuídos traz vantagens importantes. Tais como, a ilusão para o cliente que não existe réplicas/replicação, torna-se mais fácil para o programador raciocinar sobre o estado do sistema, bem como a possibilidade de adicionar mais invariantes sobre o sistema, e.g. restrições das bases de dados SQL.

Todavia, a eficiência destes algoritmos não é óptima porque requer consultar um quorum de réplicas para executar-se uma operação. Uma desvantagem que pesa bastante, em particular para aplicações desenvolvidas atualmente, é que não se consegue garantir *liveness* no sistema, o que se torna um obstáculo em *softwares* que necessitam de uma alta disponibilidade e um tempo de resposta tolerável pelo utilizador.

Anexos

	10 réplicas Sem batching	
	20 segundos de Puts Consecutivos	
Clientes	**latency (ms)**	**throughput (ops/s)**
1	5,2	192,1
10	29,9	333,2
20	41,9	476,2
30	48,9	609,3
40	52,9	752,8
50	55,9	883,5
60	59,7	997,7
70	54,8	1279,1
80	57,4	1424,6
90	69,8	1277,0
100	57,0	1777,9

	10 réplicas com batching	
	20 segundos de Puts Consecutivos	
Clientes	**latency (ms)**	**throughput (ops/s)**
1	5,1	194,9
10	21,7	458,8
20	26,8	863,8
30	31,3	947,2
40	30,6	1296,1
50	31,2	1592,2
60	34,6	1727,6
70	39,8	1763,4
80	40,5	1974,0
90	41,3	2187,3
100	38,9	2619,3

	10 réplicas	
	20 segundos de WeakGets Consecutivos	
Clientes	**latency (ms)**	**throughput (ops/s)**
1	1,15	860,5
10	1,03	3441,4
20	1,72	11558
30	3,22	9309,1
40	4,57	8754,1
50	5,92	8482,3
60	7,24	8308,15
70	8,00	8767,0
80	8,89	9061,1
90	9,90	9131,0
100	10,82	9312,2

	20 segundos de Puts Consecutivos	
#Réplicas	**latency (ms)**	**throughput (ops/s)**
3	4,435111111	225
4	6,226762321	160,3
5	7,822884013	127,6
6	9,920974155	100,6
7	16,28361858	61,35
8	21,95934066	45,5
9	30,74153846	32,5
10	37,82575758	26,4