Consenso: Paxos e Multi-Paxos

Filipe de Luna*
f.luna@campus.fct.unl.pt
MIEI, DI, FCT, UNL

Rafael Gameiro†
rr.gameiro@campus.fct.unl.pt
MIEI, DI, FCT, UNL

Rafael Sequeira‡
rm.sequeira.@campus.fct.unl.pt
MIEI, DI, FCT, UNL

ABSTRACT

O protocolo de consenso Paxos foi criado em 1989 por Leslie Lamport, e publicado oficialmente 10 anos depois, com o intuito de resolver o problema de replicação distribuída de sistemas com recurso a máquinas de estado, uma técnica sugerida também por ele próprio. Desde então, o Paxos é um algoritmo standard na indústria, utilizado por empresas como Google, Apache e Amazon. Mas, diferentes variações do Paxos têm sido propostas ao longo das últimas décadas, balanceando de formas diferentes o desempenho e tolerância a falhas.

Neste relatório são estudadas duas variações do protocolo de consenso Paxos (Paxos tradicional e Multi-Paxos), com recurso a uma máquina de estados genérica, capaz de comunicar com ambos. As respectivas implementações são especificadas sobre a forma de pseudo-código e são apresentados resultados derivados de testes experimentais realizados a fim de averiguar qual variação do algoritmo apresenta melhores resultados. Por fim, são inferidas conclusões acerca das vantagens (e desvantagens) do uso de cada uma destas versões do protocolo Paxos.

ACM Reference Format:
Filipe de Luna, Rafael Gameiro, and Rafael Sequeira. . Consenso: Paxos e Multi-Paxos. In . ACM, New York, NY, USA, 10 pages.

1 INTRODUÇÃO

Um sistema distribuído consiste num conjunto de processos distintos que trocam mensagens. [**?**] Apesar de um computador por si só poder ser visto como um sistema distribuído, o foco de mais alto-nível, como na Internet, é em diferentes máquinas a colaborar para fornecer um serviço com melhor desempenho e tolerância a falhas. A geo-replicação e tolerância a falhas são dos pontos mais importantes para um sistema distribuído do qual se espera um alcance global consistente e em tempo útil. Para alcançar ambas estas características, é necessário "replicar" o serviço, ou seja, manter o mesmo estado em todos os serviços em simultâneo.

Para tal, são utilizadas máquinas de estado, algoritmos que definem um conjunto de estados e transições entre eles. Estas máquinas realizam uma operação determinista em simultâneo que resulta num novo valor para o qual irá transitar o estado. [**?**]. Dado o ambiente de execução para sistemas distribuídos web ser naturalmente assíncrono, é necessário utilizar algoritmos específicos para manter e realizar estas transições de estado, chamados de consenso, cuja implementação tende a ser bastante complexa. O mais conhecido destes algoritmos, Paxos [**?**], foi criado em 1998 por Leslie Lamport e, desde então, já se criaram inúmeras variantes do algoritmo original [**?**].

Neste relatório iremos apresentar um estudo sobre os protocolos de consenso Paxos e a sua variante Multi-Paxos. Para tal, terá de ser criada uma implementação de uma máquina de estados, compatível com ambos estes protocolos. A implementação de todos estes componentes irá ser descrita de forma detalhada, incluindo o seu respectivo pseudo-código. Também serão apresentados os resultados dos testes de desempenho e, finalmente, será realizada uma análise que tem por objectivo principal determinar qual o protocolo que demonstra o melhor desempenho, mediante os respectivos cenários, examinando as várias métricas a serem analisadas.

Para testar e averiguar o desempenho das nossas implementações, iremos utilizar um sistema baseado na implementação de código aberto Yahoo! Cloud Serving Benchmark (YCSB). Os testes irão ser corridos no cluster computacional do Departamento de Informática da FCT NOVA, e irão consistir em simular vários clientes a fazer pedidos em simultâneo, recorrendo a instâncias do YCSB.

Após efectuar as avaliações experimentais, concluímos que existem vantagens inerentes na utilização do Paxos, na presença de um número elevado de clientes. Contudo, noutros cenários, o Multi-Paxos tende a apresentar melhor desempenho, com mais baixas latências.

O resto do documento está estruturado da seguinte forma: A secção 2 explica conceitos fundamentais à compreensão do resto do documento e um breve estudo sobre algumas variações existentes do algoritmo de consenso Paxos, sendo directamente relacionados com o nosso trabalho. Na secção 3, iremos entrar em detalhe sobre a forma como implementámos os diversos componentes que constituem o nosso sistema. Na Secção 4, são descritas as avaliações experimentais, bem como o método utilizado para a sua realização, os seus resultados e a interpretação e discussão dos mesmos. Para terminar, a Secção 5 apresenta as conclusões que foram possíveis extrair durante o decorrer da realização deste trabalho, bem como dos resultados observados nos testes.

2 TRABALHO RELACIONADO

Nesta secção iremos apresentar uma explicação acerca do protocolo de consenso Paxos e, também, algumas das suas variantes que existem mas não foram implementadas durante o decorrer deste projecto, dado estarem fora do escopo do enunciado.

O **Cheap Paxos**[**?**] consiste numa variação do Paxos original, que usa um quorum fixo de "acceptors" para permitir a execução de operações. Enquanto que, o Paxos tradicional necessita de $2F + 1$ "acceptors" e $F + 1$ réplicas, de forma a tolerar F falhas, esta variante utiliza apenas $F + 1$ "acceptors" e um conjunto com "acceptors" auxiliares, usados apenas na presença de falhas do sistema. Quando uma falha é detectada num "acceptor", este é substituído por um

*Aluno número 48425. Filipe ficou encarregado de implementar a máquina de estados.
†Aluno número 50677. Rafael ficou encarregado de implementar o protocolo Multi-Paxos.
‡Aluno número 50002. Rafael ficou encarregado de implementar o protocolo Paxos.

ASD 20/21, 2020, FCT, NOVA
December 23, 2020, Portugal.

dos "acceptors" auxiliares, de maneira a manter constantemente os $F + 1$ "acceptors", prontos para formar um quorum. É garantido que, desde que todos os "acceptors" estejam a comunicar com os lideres, não existe necessidade de "acceptors", que não participem no quorum, executem operações.

O Cheap Paxos apresenta assim uma tolerância a falhas significativamente superior, permitindo o seu correcto funcionamento com apenas $F + 1$ "acceptors".

O **Fast Paxos**[**?**] diminui o número de rondas de mensagens trocadas para permitir decisões mais rápidas do que as usadas no Paxos tradicional. Na variante original, existe pelos menos uma diferença de três rondas de mensagens, entre o momento em que uma mensagem é proposta e que passa a ser efectivamente aceite. Nesta variante, o atraso deste número de mensagens diminui para 2, mas o sistema passa a precisar de $3F + 1$ processos [**?**], tornando-se significativamente menos tolerante a falhas.

3 IMPLEMENTAÇÃO

De forma análoga ao último projecto, todos os algoritmos foram implementados com recurso à linguagem Java 11, com auxílio da biblioteca de networking "Netty" e da framework Babel, desenvolvida no laboratório da NOVA LINCS por Pedro Fouto, Pedro Ákos Costa e João Leitão.

O código base que nos foi proporcionado inclui uma aplicação servidor que é encarregue de manter um estado e aceitar pedidos de clientes, exteriores à aplicação, para ler e actualizar esse estado. Para actualizar, ler e manter o estado, teve de ser implementado, para esta aplicação, uma camada de máquina de estados e uma de consenso. A camada de máquina de estados está generalizada, mas a camada de consenso possui duas vertentes do algoritmo Paxos (Paxos tradicional e Multi-Paxos).

Em cada uma das subsecções seguintes é apresentada uma breve descrição de cada um dos três componentes por nós desenvolvidos.

3.1 Máquina de Estados

A máquina de estados está encarregue de tratar da lógica necessária para manter e actualizar o estado da aplicação. A máquina de estados é distribuída por todas as réplicas, o que permite que, com recurso à camada de consenso, seja possível ter um estado coerente entre todas as réplicas.

A máquina de estados, ao começar, se fizer parte da "membership" inicial, envia uma mensagem de "No-Op" inicial, com o papel de despoletar um comportamento alusório ao de uma "cadeia de propostas". Isto deve-se ao facto de que os protocolos baseados em Paxos funcionam em corrente. Em cada decisão da camada de consenso é proposta uma nova operação e é necessário propor algo mesmo não tendo nada para propor, uma operação intitulada de "No-Op".

Cada vez que a aplicação pretende realizar uma operação, esta é colocada num buffer, ao invés de realizá-la de imediato. Isto permite garantir que podemos ter um comportamento em parte semelhante ao do "Stubborn Delivery", em que irá ser submetida continuamente a mesma operação até esta ser realizada, resultando numa maior tolerância a falhas. O que o torna possível, é o facto de todas as operações possuírem uma hash de um número aleatório, que serve de identificador único associado, permitindo-nos saber exactamente quando é que a nossa operação é realizada.

Além de mensagens de operações a realizar pela camada de aplicação, existem mensagens de eleição de um novo líder e de adição e remoção de novas réplicas ao sistema, que também necessitam de ser ordenadas pela camada de consenso. Estas mensagens também são colocadas num buffer para garantir que são reenviadas em caso de falhas ou mudanças de líder (no caso do Multi-Paxos).

Para permitir que uma réplica consiga "apanhar" outra que esteja constantemente a enviar "No-Ops", pois esta já não tem nada para enviar, fizemos estes serem enviados apenas após um timeout que cresce de forma linear à medida que se enviam cada vez mais "No-Ops" seguidos.

Algorithm 1 State Machine

```
1:  State
2:      state; // Joining or Active
3:      membership;
4:      nextInstance;
5:      timeouts; // Keep track of timed out hosts and times
6:      stateRequests; // Keep track of state requests
7:      appRequests; // Keep track of app requests
8:      currLeader; // Current leader
9:      leaderSeqNum; // Current leader sequence number
10:
11: Init
12:     membership ⟵ InitialMembership;
13:     timeouts ⟵ {};
14:     stateRequests ⟵ {};
15:     appRequests ⟵ {};
16:     currLeader ⟵ ⊥;
17:     nextInstance ⟵ 0;
18:     If (self ∈ membership) Then
19:         state ⟵ Active;
20:         If (PaxosMode = Paxos) Then
21:             Notify(JoinedNotification, membership, nextInstance);
22:             Setup Timer DelayedNoOpTimer(FirstNoOpDelay);
23:         Else
24:             Notify(JoinedMultiNotification, membership, 0);
25:
26:     Else
27:         state ⟵ Joining;
28:         Send(StateRequest, getRandom(membership));
29:
30:
31: Upon Receive(OrderRequest, uuid, req) Do
32:     appRequests[uuid] ⟵ {req};
33:
34: Upon Receive(ForwardOrder, uuid, op) Do
35:     If (appRequests[uuid] = ⊥) Then
36:         appRequests[uuid] ⟵ {OrderRequest(uuid, op)};
37:
38:
39: Upon Receive(CurrentStateReply, state) Do
40:     Forall host ∈ stateRequests Do
41:         Send(StateReplyMessage(state, currLeader, membership, nextInstance, leaderSeqNum, host));
42:
43:     // Empty all the requests
44:     stateRequests ⟵ {};
45:
```

Algorithm 2 State Machine

```
1:  Upon Receive(DecidedNotification, uuid, opType, opId, op, host, seqN)
    Do
2:      nextInstance ⟵ nextInstance + 1;
3:      If (opType = AddReplica) Then
4:          membership ⟵ membership ⋃ {host};
5:          Call removeRedundantRequests(appRequests);
6:
7:      If (opType = RemoveReplica) Then
8:          membership ⟵ membership \ {host};
9:          Call removeRedundantRequests(appRequests);
10:
11:     If (opType = NewLeader) Then
12:         If (self = currLeader) Then
13:             Send(ForwardLeaderStateMessage, appRequests, host);
14:
15:         currLeader ⟵ host;
16:         leaderSeqNum ⟵ seqN;
17:
18:     If (opType = Read ∨ opType = Write) Then
19:         Notify(ExecuteNotification, opId, op);
20:
21:     appRequests ⟵ appRequests \ {opId};
22:     If (state = Joining) Then
23:         Return;
24:
25:     If (appRequests ≠ {}) Then
26:         reqId ⟵ getOldest(appRequests);
27:         If (PaxosMode = Paxos ∨ self = currLeader) Then
28:             Request(ProposeRequest, nextInstance, reqId, appRe-
    quests[reqId]);
29:         Else
30:             Send(ForwardOrderReqMessage, reqId, appRe-
    quests[reqId], currLeader);
31:
32:     Else
33:         If (PaxosMode = Paxos ∨ self = currLeader) Then
34:             Setup Timer DelayedNoOpTi-
    mer(computeNoOpDelay());
35:
36:
37:
38: Upon Receive(StateReply, newState, newInstance, newMembership,
    newLeader, newLeaderSeqNumber) Do
39:     Request(InstallStateRequest, newState);
40:     state ⟵ newState;
41:     nextInstance ⟵ newInstance;
42:     membership ⟵ membership ⋃ newMembership;
43:     If (PaxosMode = Paxos) Then
44:         Notify(JoinedNotification, membership, nextInstance);
45:     Else
46:         currLeader ⟵ newLeader;
47:         leaderSeqNum ⟵ newLeaderSeqNumber;
48:
49:
```

Algorithm 3 State Machine

```
1:  Upon Receive(StateRequestMessage, sender) Do
2:      uuid ⟵ generateRandomUUID();
3:      appRequests[uuid] ⟵ ReplicaRequest(uuid, AddReplica, sender);
4:
5:  Upon Receive(ForwardReplicaReqMessage, uuid, req) Do
6:      appRequests[uuid] ⟵ req;
7:
8:  Upon Receive(ForwardLeaderStateMessage, newAppRequests) Do
9:      appRequests ⟵ appRequests ⋃ newAppRequests;
10:
11: Upon Receive(OutConnectionUp, host) Do
12:     timeouts ⟵ timeouts \ host;
13:
14: Upon Receive(OutConnectionDown, host) Do
15:     uuid ⟵ generateRandomUUID();
16:     appRequests[uuid] ⟵ ReplicaRequest(uuid, RemoveReplica,
    host);
17:
18: Upon Receive(OutConnectionFailed, host) Do
19:     If (host ∉ membership) Then
20:         Return;
21:
22:     If (host ∉ timeouts) Then
23:         timeouts[host] ⟵ 0;
24:     Else
25:         timeouts[host] ⟵ timeouts[host] + 1;
26:
27:     If (timeouts[host) Then < MaxTimeoutTries]
28:         Setup Timer TimeoutTimer(host, TimeoutInterval);
29:     Else
30:         uuid ⟵ generateRandomUUID();
31:         appRequests[uuid] ⟵ ReplicaRequest(uuid, RemoveReplica,
    host);
32:
33:
```

3.2 Paxos

O Paxos foi apresentado pela primeira vez em 1998 por Leslie Lamport [**?**] com objectivo de resolver o problema de consenso num sistema distribuído. A noção de consenso é o alicerce de um sistema baseado na replicação de máquinas de estado. Este tipo de sistemas consistem em ter um estado replicado em todos os nós do sistema, nos quais são executadas operações determinísticas de forma ordenada, com o intuito de aumentar a disponibilidade e tolerância a falhas. Para alcançar esta ordem, as diversas partes utilizam um sistema de rondas que tem como objectivo chegar a um acordo sobre qual a operação a ser executada nessa ronda por todos os participantes.

Numa das vertentes exploradas neste trabalho - o Paxos tradicional - existe um número arbitrário de réplicas que tentam propor, de forma concorrente, uma operação a ser executada. Estas propostas são divididas em duas fases distintas: A primeira - a fase de "prepare" - começa com o envio de um número de sequência por parte de cada réplica. A segunda ocorre quando uma réplica consegue reunir uma maioria de respostas a estas mensagens, encontrando-se em condições de propor uma operação. A operação proposta, nesta segunda fase, pode ser a sua operação inicial ou, caso alguma das outras réplicas já tenha aceite uma proposta no

passado, o valor desta última. De seguida, as réplicas chegam a um acordo após a proposta de uma operação conseguir aprovação da maioria das réplicas, terminando assim o algoritmo.

Dada a limitação inerente ao Paxos de apenas permitir a decisão de um valor por execução, para a nossa implementação criámos uma camada de abstracção sobre o Paxos que gere as diferentes instâncias à custa do conceito de estado. Um estado está sempre associado a uma instância e agrega todas as variáveis referentes à mesma. Duas instâncias, e por consequência dois estados, são sempre totalmente independentes entre si. Apesar disto, as decisões associadas a cada uma das instâncias são tomadas de forma sequencial. Isto acontece pois, para tomar uma decisão numa determinada instância, é necessário conhecer todas as réplicas participantes, e, uma vez que as operações podem alterar esta "membership", só é possível ter a certeza da sua composição após a instância imediatamente anterior terminar. No entanto, isto não implica que uma réplica não possa participar em instâncias que não a sua instância actual.

Para permitir a progressão de réplicas que se encontrem atrasadas relativamente à replica em questão, esta deve responder a mensagens de uma instância "passada". E, de forma a permitir o progresso de réplicas avançadas relativamente a esta, também foram implementados mecanismos de resposta a instâncias "futuras". Estes mecanismos permitem a uma réplica participar em todos os eventos de uma ronda deste tipo, desde que não seja necessário recorrer à "membership" para o fazer.

Abaixo é apresentado o pseudo-código relativo à implementação do Paxos no nosso projecto.

Algorithm 4 Paxos

```
1:  Upon Receive(PREPARE, p, ins, seqNum) Do
2:      // if instance does not exist yet
3:      If ( ins ∉ createdInstances ) Then
4:          createNewInstance(ins);
5:
6:      If (seqNum > highestPrepare[ins]) Then
7:          highestPrepare[ins] ⟵ seqNum;
8:          Send(ACCEPT-OK, p, instance, highestAccept[ins], acceptedValue[ins]);
9:
10:
11: Upon Receive(PREPARE-OK, p, ins, seqNum, value) Do
12:     If (seqNum < instance ∨ lastPrepare[ins] ≠ seqNum ∨ (membership[ins] ≠ ⊥ ∧ #prepareOks[ins] > #membership/2)) Then
13:         Return;
14:
15:     If (value ≠ ⊥ ∧ value > proposeValue[ins]) Then
16:         proposeValue[ins] ⟵ value;
17:
18:     prepareOks[ins] ⟵ prepareOks[ins] ⋃ {PREPARE-OK,ins , seqNum, value};
19:     If (membership[ins] ≠ ⊥ ∧ #prepareOks[ins] > #membership/2)) Then
20:         Forall m ∈ membership[ins] Do
21:             Send(ACCEPT, m, instance, lastPrepare[instance], proposeValue[ins]);
22:
23:
24:
```

Algorithm 5 Paxos

```
1:  Interface
2:      Indications
3:          Propose(instance,opId,op);
4:
5:
6:  State
7:      // current instance
8:      instance;
9:      // map with peers in a membership for each instance
10:     membership;
11:     // map with the peer that received an accept
12:     // ok before the membership was defined for each instance
13:     acceptedPeer;
14:     // map with current sequence numbers in each instance
15:     sn;
16:     // map with highest accept sequence number accepted in each instance
17:     highestAccept;
18:     // map with highest prepare sequence number accepted in each instance
19:     highestPrepare;
20:     // map with highest learned sequence
21:     // number accepted in each instance
22:     highestLearned;
23:     // map with a sequence number used
24:     // in the last prepare message sent in each instance
25:     lastPrepare;
26:     // map with sets of prepare ok messages received in each instance
27:     prepareOks;
28:     // map with sets of accept ok messages received in each instance
29:     acceptOks;
30:     // map with the value that will be proposed in each instance
31:     proposeValue;
32:     // map with the accepted value in each instance
33:     acceptedValue;
34:     // map with the decided value in each instance
35:     decidedValue;
36:     // set with all created instances
37:     createdInstances;
38:
39: Init
40:     instance ⟵ -1;
41:     membership ⟵ [;]
42:     acceptedPeer ⟵ [;]
43:     sn ⟵ [;]
44:     highestAccept ⟵ [;]
45:     highestPrepare ⟵ [;]
46:     highestLearned ⟵ [;]
47:     lastPrepare ⟵ [;]
48:     prepareOks ⟵ [;]
49:     acceptOks ⟵ [;]
50:     proposeValue ⟵ [;]
51:     acceptedValue ⟵ [;]
52:     decidedValue ⟵ [;]
53:     createdInstances ⟵ {};
54:
```

Algorithm 6 Paxos

```
1:  Upon Receive(ACCEPT, p, ins, seqNum, val) Do
2:      // if instance does not exist yet
3:      If ( ins ∉ createdInstances ) Then
4:          createNewInstance(ins);
5:
6:      If (seqNum > highestPrepare[ins]) Then
7:          highestPrepare[ins] ⟵ seqNum;
8:          highestAccept[ins] ⟵ seqNum;
9:          acceptedValue[ins] ⟵ value;
10:         If ( membership[ins] ≠ ⊥ ) Then
11:             Forall m ∈ membership[ins] Do
12:                 Send(ACCEPT-OK, m, instance, sn[instance], acceptedValue[ins]);
13:
14:         Else
15:             acceptedPeer[ins] ⟵ p;
16:             Send(ACCEPT-OK, p, instance, sn[instance], acceptedValue[ins]);
17:
18:
19:
20: Upon Receive(ACCEPT-OK, ins, seqNum, value) Do
21:     // if instance does not exist yet
22:     If ( ins ∉ createdInstances ) Then
23:         createNewInstance(ins);
24:
25:     If (seqNum < highestLearned[ins] ∨ decidedValue[ins] ≠ ⊥ ) Then
26:         Return;
27:
28:     If (seqNum > highestLearned[ins]) Then
29:         highestLearned[ins] ⟵ seqNum;
        // delete all accept oks received so far
30:         acceptOks[ins] ⟵ {};
31:
32:     acceptOks[ins] ⟵ acceptOks[ins] ⋃ {ACCEPT-OK, ins, seqNum, value};
33:     If (membership[ins] ≠ ⊥ ∧ #acceptOks[ins] > #membership/2) Then
34:         Cancel Timer TIMER;
35:         decidedValue[ins] ⟵ value;
36:         Notify(DecidedNotification, ins, value);
37:
38:
```

Algorithm 7 Paxos

```
1:  Upon Propose(ins, value) Do
2:      instance ⟵ ins;
3:      // if instance does not exist yet
4:      If ( ins ∉ createdInstances ) Then
5:          createNewInstance(ins);
6:
7:      If ( membership[ins] = ⊥ ) Then
8:          membership[instance] ⟵ membership[instance-1];
9:          // generate unique first sequence number
10:         sn[ins] ⟵ identifier + #membership[ins] * pickRandom(100);
11:         If ( highestAccept[ins] > 0) Then
12:             Forall m ∈ membership[in] ∧ m ≠ acceptedPeer[ins] Do
13:                 Send(ACCEPT-OK, ins, highestAccept[ins], acceptedValue[ins]);
14:
15:
16:
17:     proposeValue[ins] ⟵ value;
18:     sendPrepare(ins);
19:
20: Upon Timer Timer(TIMER) Do
21:     prepareOks[instance] ⟵ {};
22:     // must be higher than any sequence number seen so far
23:     sn[ins] ⟵ pickNewSequenceNumber();
24:     sendPrepare(ins);
25:
26: Procedure sendPrepare(instance)
27:     lastPrepare[instance] ⟵ sn[instance];
28:     Setup Timer (TIMER, timeout);
29:     Forall m ∈ membership[ins] Do
30:         Send(PREPARE, instance, sn[instance]);
31:
32:
33: Upon Receive(JoinedNotification, ins, peers) Do
34:     // init everything related to an instance
35:     createNewInstance(ins);
36:     membership[ins] ⟵ peers;
37:     // generate unique first sequence number
38:     sn[ins] ⟵ identifier + #membership[ins] * pickRandom(100);
39:
40: Upon Receive(MembershipChangeNotification, opType, p) Do
41:     If (opType = REMOVE_REPLICA) Then
42:         membership[instance] ⟵ membership[instance-1] \ p;
43:     Else If (opType = ADD_REPLICA) Then
44:         membership[instance] ⟵ membership[instance-1] ⋃ p;
45:
46:
```

3.3 Multi-Paxos

O Multi-Paxos [**?**], inicialmente apresentado no primeiro paper do Paxos [**?**], é possivelmente das variantes de Paxos mais utilizadas actualmente. Este protocolo destaca-se do seu original, não só por ser capaz de atingir mais rapidamente o consenso, como também pelo facto de que permite uma redução da troca de mensagens entre as replicas pertencentes ao sistema.

O modo de operação deste algoritmo consiste em eleger um "distinguish proposer" que possibilita uma ordem na execução das

operações mais estrita, uma vez que só esta réplica é que pode propor valores. Isto permite, na prática, retirar uma ronda de mensagens usada no Paxos simples para propor valores. Esta mesma ronda é reutilizada para quando ocorre eleição de um líder.

A eleição de um líder, denominada "Leader Election", começa com uma troca de mensagens entre réplicas em que cada uma se propõe ser líder ao enviar o seu número de sequência, sendo elevada a essa posição a que enviar o número mais alto. Apenas a réplica que apresenta o número de sequencia mais elevado será eleita como líder. A partir desse momento, todas as rondas de mensagens seguintes até à próxima eleição, usarão como número de sequência o valor proposto pela réplica líder. É importante realçar, que a nossa implementação do Multi-Paxos diverge ligeiramente da apresentada no paper original. Neste paper, o número de sequência usado após uma eleição é zero, contudo, essa decisão pode gerar alguma controvérsia, o que nos levou a decidir utilizar o número de sequência usado pela réplica durante a fase de eleição, tal como nos fora apresentado durante as aulas.

As restantes rondas de mensagens a seguir à eleição, consistem em executar uma sequência de mensagens "Accept-Accept-OK" de forma a decidir os valores propostos pelos clientes. Caso durante um determinado intervalo de tempo não haja pedidos de clientes, o algoritmo irá executar operações "No-Op", de forma a manter a "leadership" actual.

De forma a garantir que o protocolo consegue progredir, cada uma das réplicas tem um temporizador que é activado quando o líder, durante um determinado intervalo de tempo, não envia nenhum pedido. Nesse momento, é feita uma nova eleição, resultando num novo líder, com um novo número de sequencia.

Da mesma maneira que as réplicas possuem um temporizador para iniciar uma nova eleição, o líder também possui um temporizador que é activado quando, durante uma ronda de " Accept-Accept-OK", não recebe uma maioria de respostas. Nesta situação, uma nova eleição é desencadeada e, consequentemente, um novo líder é eleito.

Algorithm 8 Multi-Paxos

```
1:  Interface
2:      Indications
3:          Propose(instance,opId,op);
4:
5:
6:  State
7:      // current instance
8:      instance;
9:      // own sequence number
10:     currentPrepare;
11:     // highest sequence number registered
12:     highestPrepare;
13:     // set with peers in membership
14:     membership;
15:     // set of peers that received the same request during a leader election
16:     prepareRequests;
17:     // set of peers that received the same request during an normal instance
18:     acceptRequests;
19:     // registry os messages decided per instance
20:     log;
21:     // set of messages to accept after the leader election happen
22:     msgToAccept;
23:     // messages that must be resended after the resend timer was activated
24:     resendMessages;
25:     // current the leader
26:     leader;
27:     // defines when a leader election is happening
28:     leaderElection;
29:
30: Init
31:     instance ⟵ -1;
32:     currentPrepare ⟵ -1;
33:     highestPrepare ⟵ -1;
34:     membership ⟵ {};
35:     prepareRequests ⟵ {};
36:     acceptRequests ⟵ {};
37:     log ⟵ [;]
38:     msgToAccept ⟵ {};
39:     resendMessages ⟵ {};
40:     leader ⟵ ⊥;
41:     Setup Timer (START, timeout_1);
42:
43: Upon Prepare() Do
44:     Forall currentPrepare ≤ highestPrepare Do
45:         currentPrepare ⟵ currentPrepare ⋃ highestPrepare;
46:
47:     highestPrepare ⟵ currentPrepare;
48:     leaderElection ⟵ true;
49:     acceptRequests ⟵ {};
50:     prepareRequests ⟵ {myself};
51:     Forall p ∈ peers ∧ p ≠ myself Do
52:         Send(PREPARE, p, myself, instance, highestPrepare);
53:
54:     Setup Timer (RESEND, instance, EMPTY_ID, EMPTY_OP, timeout_2);
55:
```

Algorithm 9 Multi-Paxos

```
1:  Upon addMessagesToAccept() Do
2:      Forall size(resendMessages) > 0 Do
3:          message ⟵ first(resendMessages);
4:          If (message ∉ msgToAccept) Then
5:              msgToAccept ⟵ msgToAccept ⋃ message;
6:
7:
8:
9:  Upon nextMessageToAccept() Do
10:     message ⟵ first(msgToAccept);
11:     If (message ≠ ⊥ ∧ msg.getType = PROPOSE_MESSAGE) Then
12:         messageToSend ⟵ (ACCEPT_MESSAGE, message.getInstance(), highestPrepare, message.getOpId(), message.getOp());
13:         Forall p ∈ membership Do
14:             Send(PROPOSE, p, myself, message, highestPrepare);
15:
16:         Setup Timer (RESEND, instance, EMPTY_ID, EMPTY_OP, $timeout_2$);
17:     Else If (#msgToAccept > 0) Then
18:         nextMessageToAccept();
19:
20:
21: Upon Receive(PREPARE, p, ins, seqNum) Do
22:     If (instance ≥ 0 ∧ seqNum > highestPrepare) Then
23:         highestPrepare ⟵ seqNum;
24:         leader ⟵ p;
25:         leaderElection ⟵ false;
26:         pMsg ⟵ ⊥;
27:         If (∃ p ∉ log) Then
28:             instance ⟵ ins;
29:             pMsg ⟵ (ins, seqNum, {});
30:         Else
31:             pMsg ⟵ (ins, seqNum, getLog(ins));
32:
33:         // must send back the values accepted in other N iterations(?)
34:         Send(PREPARE-OK, p, myself, ins, seqNum);
35:         Setup Timer (LEADER, instance + 1, $timeout_3$);
36:
37:
38: Upon Receive(PREPARE-OK, p, ins, seqNum) Do
39:     If (instance = ins ∧ p ∉ prepareRequests) Then
40:         prepareRequests ⟵ prepareRequests ⋃ p;
41:         If (#prepareRequests > #membership/2) Then
42:             leader ⟵ myself;
43:             leaderElection ⟵ false;
44:             prepareRequests ⟵ {};
45:             Notify(DeliverNotification, ins, highestPrepare, getLeaderOperation());
46:             addToMessagesToAccept();
47:             If (#msgToAccept > 0) Then
48:                 nextMessageToAccept();
49:
50:
51:
52:
```

Algorithm 10 Multi-Paxos

```
1:  Upon Receive(ACCEPT, p, ins, seqNum, opId, op) Do
2:      If (instance < 0 ∨ seqNum < highestPrepare ∨ leaderElection) Then
3:          Return;
4:
5:      If (ins > instance + 1) Then
6:          leader ⟵ p;
7:          highestPrepare ⟵ seqNum;
8:          uponPrepareRequest();
9:          Return;
10:
11:     If (seqNum > highestPrepare) Then
12:         leader ⟵ p;
13:
14:     If (ins ∉ log) Then instance ∧ ins
15:         Notify(DecidedNotification, ins, highestPrepare, getLeaderOperation());
16:
17:     acceptRequests ⟵ {};
18:     acceptRequests ⟵ {myself};
19:     Forall p ∈ peers ∧ p ≠ myself Do
20:         Send(ACCEPT-OK, p, myself, instance, highestPrepare, opId, op);
21:
22:
23: Upon Receive(ACCEPT-OK, p, ins, seqNum, opId, op) Do
24:     If (instance = ins ∧ p ∉ acceptRequests ∧ ¬leaderElection) Then
25:         acceptRequests ⟵ acceptRequests ⋃ p;
26:         If (#acceptRequests > #membership/2) Then
27:             acceptRequests ⟵ {};
28:             If (instance ∉ log) Then
29:                 Notify(DecidedNotification, ins, opId, op);
30:                 log ⟵ log ⋃ (REPLAYACCEPTMESSAGE, instance, PROPOSE_MESSAGE, opId, op, myself);
31:
32:             Setup Timer (LEADER, instance + 1, $timeout_3$);
33:             If (leader ≠ myself ∧ msgToAccept > 0) Then
34:                 nextMessageToAccept();
35:
36:
37:
38:
```

Algorithm 11 Multi-Paxos

```
1:  Procedure computeCurrentPrepareWLeader(peers, lead)
2:      membership ⟵ membership \ myself;
3:      membership ⟵ sortByHost(membership);
4:      pos ⟵ indexOf(membership);
5:      distance ⟵ size(membership) - pos;
6:      currentPrepare ⟵ highestPrepare + distance + 1;
7:
8:  Procedure computePrepareNumbers(peers)
9:      membership ⟵ sortByHost(membership);
10:     pos ⟵ indexOf(membership);
11:     currentPrepare ⟵ pos;
12:     highestPrepare ⟵ pos;
13:
14: Upon Receive(JoinedMultiNotification, ins, seq, peers, lead) Do
15:     isntance ⟵ ins;
16:     membership ⟵ peers;
17:     leader ⟵ lead;
18:     If (leader = ⊥) Then
19:         computePrepareNumbers(peers);
20:         startTimer ⟵ getRandomNumber(min, max);
21:         Setup Timer (START, timeout_1);
22:     Else
23:         highestPrepare ⟵ seq;
24:         computeCurrentPrepareWLeader(peers, lead);
25:
26:
27: Upon Receive(MembershipChangeNotification, opType, p) Do
28:     If (opType = REMOVE_REPLICA) Then
29:         membership ⟵ membership \ p;
30:     Else If (opType = ADD_REPLICA) Then
31:         membership ⟵ membership ⋃ p;
32:
33:
```

4 AVALIAÇÃO EXPERIMENTAL

Nesta secção iremos expor as avaliações experimentais efectuadas para verificar e validar as implementações feitas. Cada teste efectuado consistiu na utilização de um dos protocolos de consenso baseados em Paxos, implementados na secção 3, e de um cliente de benchmark, que nos foi disponibilizado pela docência.

4.1 Metodologia de Teste

Os testes realizados consistiram em correr, num cluster computacional, aplicações de servidor e cliente. Cada aplicação servidor consiste numa "Hash App", a aplicação encarregue de receber os pedidos dos clientes, uma máquina de estados e um algoritmo de consenso (Paxos ou Multi-Paxos). A aplicação cliente, foi-nos fornecida e consiste num programa que lança vários threads que realizam pedidos de forma concorrente a um dos servidores replicados. Foram corridas, para todos os testes, 3 aplicações servidor e 2 aplicações cliente.

Foi-nos também fornecido um script encarregue de lançar as réplicas e clientes necessários, tendo em conta quaisquer nós que tenhamos reservado no cluster computacional. Este script, da autoria do docente da unidade curricular Pedro Fouto, foi apenas levemente modificado por nós para permitir, adicionalmente, realizar vários testes com diferentes configurações de forma autónoma. O parâmetro que era necessário variar, era o respectivo ao número de processos em cada aplicação cliente, ou seja, o número "real" de clientes a fazer pedidos concorrentemente.

4.2 Métricas de teste

As métricas de teste que foram utilizadas para este projecto, a fim de interpretar o desempenho das diferentes vertentes do algoritmo de consenso Paxos, são: O número de operações por segundo ("throughput") e a latência dos pedidos. Ao juntar aos valores de ambas estas métricas o número de clientes com que foi configurado o teste, será possível extrapolar o ponto em que o sistema se satura e o desempenho começa a decair.

4.3 Hardware

As máquinas que escolhemos reservar no cluster computacional para realizar os testes, possuíam todas as mesmas especificações, que consistiam em 2 processadores AMD Opteron 2376 com 16 GB de memória RAM e 2 canais de 1 Gbps. Escolhemos estas máquinas não só por existirem em grande número, permitindo uma homogeneidade no sistema, que por sua vez evita largas discrepâncias nos resultados dos testes, mantendo a coerência dos mesmos, mas também por possuírem um hardware mais fraco, que se torna mais fácil de chegar a um ponto de estrangulamento.

Todas as máquinas corriam uma distribuição Ubuntu LTS. As máquinas estavam hospedadas num cluster de computação pertencente ao Departamento de Informática da FCT NOVA[**?**], cujo acesso nos foi proporcionado pela equipa docente com o intuito de correr as experiências necessárias à realização deste relatório. As máquinas utilizadas pelo cluster eram geridas pelo software de gestão de clusters *OAR*[**?**].

4.4 Configurações

4.4.1 ***State Machine:*** Após diversos testes, decidimos utilizar um valor inicial de 10 ms para o temporizador que define o tempo de espera até ser enviado um "No-Op". Este valor incrementará por 2 ms a cada "No-Op" seguido. até um máximo de 50 ms. Este valor apresentou uma excelente relação entre desempenho e tempo de recuperação de um nó atrasado.

4.4.2 ***Paxos:*** Para o Paxos, foi configurado um temporizador com o valor de 150 ms responsável por aumentar o número de sequência, caso um pedido de "Prepare" e "Accept" não obtenha resposta.

4.4.3 ***Multi-Paxos:*** De forma a manter consistência com os valores de temporizadores definidos no Paxos, o Multi-Paxos utilizou um tempo de 200 ms para activar o "Resend Timer". O "Leader Timer" por sua vez utilizou um valor mais alto do que definido pelo intervalo máximo de "No-Ops", sendo este 700 ms. Para evitar que todas as instâncias pudessem começar ao mesmo tempo e assim retardar o processo de eleição de um líder, gerou-se um número aleatório entre 3000 e 5000, com o objectivo de possibilitar um grande intervalo de valores e diminuir a probabilidade de existirem colisões.

4.5 Resultados

Os resultados foram obtidos através do output gerado pela aplicação de cliente, e extrapolados para gráficos referentes a cada um

dos algoritmos de consenso - Paxos e Multi-Paxos. As métricas de latência e throughput permitem-nos obter o limite de máximo de clientes concorrentes que o sistema consegue suportar com 3 réplicas.

Os gráficos obtidos contém o throughput e latência alcançada ao correr cada um dos algoritmos de consenso com 6, 12 24 32 64 e 128 clientes. Estes clientes referem-se ao número de threads que estariam a executar operações de forma concorrente, distribuídos pelas aplicações cliente, durante o decorrer do teste.

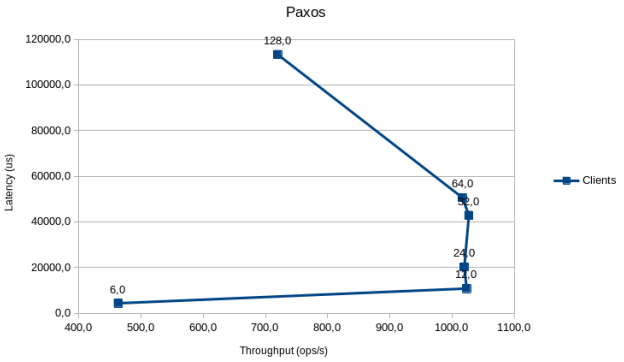

Figure 1: Desempenho do algoritmo Paxos com um número crescente de clientes.

Como podemos verificar no gráfico 1, o desempenho cresceu substancialmente, triplicando, sem um aumento significativo de latência, ao aumentar de 6 para 12, o número de clientes a realizar pedidos. Mas, estes 12 aparentam ser o número máximo de clientes que o sistema consegue suportar, sem começar a ser perceptível uma deterioração do seu desempenho. Esta deterioração, até aos 64 clientes, é apenas reflectida na latência de resposta a pedidos dos clientes, mantendo o "throughput" máximo.

Ao chegar aos 128 clientes, isto já não se verifica, com uma deterioração linear tanto do "throughput" como da latência. Isto deve-se ao facto de a carga crescente induzida no sistema resultar num crescimento exponencial do tempo de decisão do algoritmo de consenso. A causa deste comportamento situa-se no mecanismo de ordenamento de operações propostas pelo cliente, em que cada réplica está encarregue de propor uma operação a fim de concluir uma instância do algoritmo de consenso. Uma vez que cada réplica procura que o algoritmo execute a operação proposta por si, devido à maior afluência de operações, irá ser necessário um maior intervalo de tempo até que o consenso por instância seja atingido.

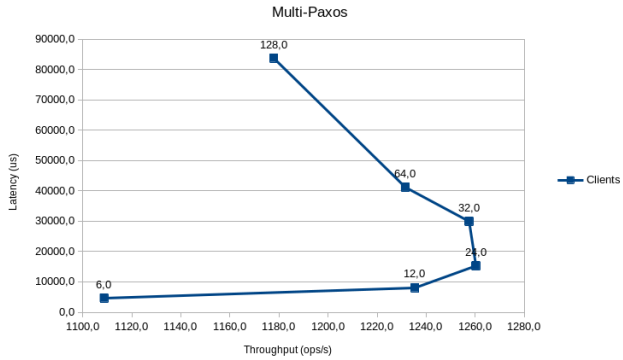

Figure 2: Desempenho do algoritmo Multi-Paxos com um número crescente de clientes.

Para o Multi-Paxos, é possível examinar os seus resultados da avaliação experimental no gráfico 2. A forma resultante da disposição dos valores do gráfico apresentado, assemelha-se à do gráfico referente ao Paxos, dado que, apesar das suas particularidades, o algoritmo ainda mantém a maior parte da lógica. Contudo, o Multi-Paxos apresenta um melhor desempenho que o Paxos, alcançando "throughputs" mais altos, aliados a uma latência mais baixa. Porém, dado que todo a carga do sistema incide num único nó (o líder), a saturação ocorre mais cedo, aos 32 clientes e apresenta um crescimento linear semelhante ao visto anteriormente no gráfico referente ao Paxos. Esta ausência da característica de "load-balancing", traduz-se numa capacidade reduzida do sistema em conseguir lidar com o aumento elevado pedido propostos pelo cliente, como pudemos verificar no gráfico apresentado. O sistema, ao redireccionar todos os pedidos recebidos para o líder, faz com que este seja o ponto de garrote do sistema. Esse factor, em conjunto com o intervalo persistente, inerente ao protocolo de consenso e referente a cada instância, leva a que eventualmente o sistema atinja a saturação mais cedo.

5 CONCLUSÃO

Neste relatório fizemos um conjunto de avaliações experimentais em algoritmos ja existentes, sobre as nossas implementações dos algoritmos de consenso, Paxos e Multi-Paxos. Para cada algoritmo, verificámos a deterioração do desempenho com o aumento de clientes.

Pudemos concluir que o algoritmo Multi-Paxos, comparativamente ao Paxos simples, apesar de apresentar um melhor desempenho, com um número mais elevado de operações por segundo, atinge o seu ponto de estrangulamento mais cedo. Tendo em conta que a métrica respectiva à tolerância a falhas encontra-se fora do escopo desta avaliação experimental, conseguimos verificar que o algoritmo Paxos, apesar de apresentar um desempenho inferior, torna-se uma escolha mais apropriada para um sistema com um elevado número de clientes.

Apesar de, o Multi-Paxos ter a vantagem significativa de aliviar a exigências na largura de banda do sistema, por reduzir drasticamente o numero de mensagens por sistema, durante os nossos

testes, verificámos que a aplicação esgotou o poder de processamento antes de ser possível esgotar a largura de banda dos canais. Isto deve-se, em grande parte, ao facto de que as mensagens trocadas eram de muito pequena dimensão.