Implementation and Experimental Measurements of State Machine Replication using Paxos and Multi-Paxos

Rodrigo Faria Lopes*
rfa.lopes@campus.fct.unl.pt
MIEI, DI, FCT, UNL

Miguel Candeias†
mb.candeias@campus.fct.unl.pt
MIEI, DI, FCT, UNL

Salvador Rosa Mendes‡
sr.mendes@campus.fct.unl.pt
MIEI, DI, FCT, UNL

ABSTRACT

Este artigo tem como objetivo a implementação e comparação dos algoritmos Paxos [1] [3] [4] [5] e Multi-Paxos, sendos estes utilizados como instâncias da Máquina de estados que alimenta a HashApp, sendo esta responsável por falar com os clientes.

Neste artigo iremos falar sobre as implementações dos algoritmos acima descritos, explicando o seu funcionamento e a forma como foram implementados. Por fim iremos apresentar algumas medidas de comparação entre os algoritmos.

ACM Reference Format:
Rodrigo Faria Lopes, Miguel Candeias, and Salvador Rosa Mendes. 2020. Implementation and Experimental Measurements of State Machine Replication using Paxos and Multi-Paxos. In *The Projects of ASD - first delivery, 2020, Faculdade de Ciências e Tecnologia, NOVA University of Lisbon, Portugal*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

No contexto deste segundo projeto de ASD, foi implementado e analisado um algoritmo de *State Machine Replication*, e dois protocolos de *Agreement* denominados por Paxos e o Multi Paxos.

O algoritmo de *State Machine Replication* é importante, porque permite a criação de serviços distribuidos, com propriedades de tolerância a falhas, cujo o seu papel é tratar dos pedidos efetuados pelos clientes, e efetuar a gestão do estado geral do sistema, utilizando um algoritmo do tipo *Agreement*.

Os algoritmos do tipo *agreement*, auxiliam assim a *State Machine Replication*, chegando em conjunto entre todas as instâncias do algoritmo *agreement*, a um consenso.

O artigo divide-se então na apresentação dos protocolos implementados, seguidamente da seccção de implementação onde se explica pormenores de implementação destes, e também o pseudo-codigo e uma explicação do mesmo. Finalmente, iremos expor os problemas encontrados na secção de experimentação.

2 RELATED WORK

2.1 State Machine

State Machine Replication é um método para implementar replicação de servidores mantendo a ordem das interações dos clientes e oferecendo tolerância de falhas. Para a State Machine Replication funcionar as State Machines que integram o sistema têm de ser deterministas, sendo assim todas têm de começar com o mesmo estado inicial e o resultado produzido por um pedido tem de ser o mesmo em todas as máquinas, ou seja, a partir de um estado X e com uma ação Y o resultado será sempre o mesmo, em caso de dúvida acerca do resultado por parte da aplicação utiliza-se a maioria. Uma State Machine é composta por um conjunto de estados, um conjunto de inputs, um conjunto de outputs, um função de transição de estado (State x Input -> State'), uma função de output (State x Input -> Output) e um estado inicial. A ordem das operações a executar é dada pela State Machine e depende da ordem de chegada dos pedidos e, no nosso projeto, a confirmação da operação a executar é dada por um agreement protocol que será o Paxos ou o Multi-Paxos.

2.2 Paxos

O Paxos é um algoritmo de concordância, onde o principal objetivo é que todas as replicas decidam o mesmo valor. Para isso o algoritmo pode dividir-se em três entidades diferentes, os Proposers, Acceptors e os Learners, tendo cada um destes uma missão dentro do protocolo.

2.2.1 Proposers. Ao receberem o valor requerido pelo cliente iram enviar a todos os Acceptors umas mensagem de Prepare, ficando à espera depois de um reply chamado Prepare_Ok. Após receber os Prepare_Ok necessários, passará ao envio da mensagem de Accept que por sua vez poderá também ter que esperar pelo Accept_Ok dos Acceptores de modo a que quando receba um Quorum de mensagens minimo para o mesmo valor, consiga decidir essa valor com a garantia de que todas as replicas converjam para a mesma decisão.

2.2.2 Acceptors. Estes ao receberem uma mensagem de Prepare dos Proposers, vão verificar se o número de sequência enviado por este é maior ao número de sequência visto anteriormente, e caso esta verificação não se verifique, este apenas ignorará a mensagem do Proposer. Caso contrário, o Acceptor enviará a mensagem de Accept_Ok ao Proposer avançando mais um passo no protocolo.

O mesmo se sucede à recepção da mensagem de Accept, onde o Acceptor volta a comprar o número de sequência de forma semelhante enviando neste caso um Accept_Ok.

2.2.3 Learners. No caso dos Learners existe duas abordagens chave: Distinguished Proposer, em que o Proposer quando recebe a maioria de Accept_OK manda Decided para todos os Learners, ou o Regular em que os Acceptors mandam os Accept_Ok para todas as replicas, e quando um Learner observa um Quorum de Accepts, enviam a decisão para o cliente.

2.3 Multi Paxos

O estudo deste algoritmo, teve como base os materiais leccionados nas aulas. O Multi Paxos é um algoritmo do tipo *agreement*,

*Student number 50435. Rodrigo foi responsável pela implementação do Paxos.
†Student number 50647. Miguel foi o responsável pela implementação do Multi-Paxos
‡Student number 50503. Salvador foi o responsável pela implementação da State Machine.

ASD20/21, 2020, FCT, NOVA University of Lisbon, Portugal

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

cujo o seu objetivo é levar varias instâncias deste algoritmo, distribuídas, a chegarem a um consenso acerca de uma dada operação.

No caso do Multi Paxos, este é uma melhoria relativamente ao algoritmo Paxos, previamente descrito. Esta melhoria deve-se ao facto de conseguir garantir a propriedade de *liveness*, que só por si o Paxos não consegue resolver. Trata-se de um problema, no caso de quando um processo invoca uma *prepare operation*, e devido a falhas, nunca se conseguirá reunir uma maioria de *quorums*, podendo assim nunca se chegar a um resultado. O Multi Paxos evita este problema, recorrendo à eleição de um processo Lider, sendo este o responsável pelo tratamento de todos os pedidos efetuados pela *State Machine*. Sendo assim, no Multi Paxos, o processo lider não necessita de efetuar *prepare operations*, podendo assim disseminar o *AcceptOk* pelo resto das instâncias do Multi Paxos.

3 IMPLEMENTATION

3.1 State Machine

Com o auxilio do material leccionado acerca de State Machine Replication foi possível a implementação do mesmo, tendo tido em especial atenção o pseudo-código do Uniform Concensus. Uma melhoria realizada nas State Machines foi a criação de um *buffer* para operações que chegariam às réplicas enquanto estas ainda estava a executar um pedido ao paxos ou quando ainda não estavam disponíveis para funcionamento por se estarem a juntar à *membership*. Esse *buffer* de operações é composto por uma *queue* de UUID de operações de forma a manter a ordem de entrada e um mapa que relaciona cada UUID da *queue* com um objecto operação. As operações que eram aqui registadas eram posteriormente chamadas quando a máquina acaba-se de se juntar à rede e sempre que a máquina tomava uma decisão e tinha de escolher qual a próxima a ser executada.

O Algorithm 1, 2 e 3 representam a nossa implementação da State Machine na *framework* Babel.

Algorithm 1 State Machine

```
1:  function Init
2:      nextInstance ← 0
3:      executedOps ← 0
4:      data ← Map
5:      agreement ← props.Agreement
6:      operationQueue ← Queue
7:      opExecuting ← null
8:      currentOp ← null
9:      cumulativeHash ←
10:     membership ← initialMembership
11:     opsToBe ← Queue
12:     operationMap ← Map
13:     hasState ← False
14:     hasMembership ← False
15:     if self ∈ membership then
16:         state ← ACTIVE
17:         for host ∈ membership do
18:             openConnection(host)
19:             Trigger send(JoinedNotification, membership,
    nextInstance)
20:         end for
21:     else
22:         state ← JOINING
23:         for host ∈ membership do
24:             openConnection(host)
25:             Trigger send(JoinedNotification, membership,
    nextInstance)
26:         end for
27:         sendMessage(RequestMembership, membership[0])
28:         sendRequest(CurrentStateRequest, protocolId)
29:     end if
30: end function

31: function uponStateReply( CurrentStateReply, Source-
    Proto )
32:     tempState ← CurrentStateReply.getState()
33:     executedOps ← tempState.getExecutedOps()
34:     cumulativeHash ← tempState.getCumulativeHash()
35:     data ← tempState.getData()
36:     hasState ← True
37:     if hasMembership == True and hasState == True then
38:         state ← ACTIVE
39:         Trigger send(JoinedNotification, membership, nex-
    tInstance)
40:         if opsToBe.size() > 0 then
41:             Call uponOrderRequest(OrderRequest, proto-
    colId)
42:         end if
43:     end if
44: end function
```

3.2 Agreement

3.2.1 Paxos. Na implementação do Algoritmo Paxos, inspiramos-nos no pseudo-código fornecido das aulas teóricas tendo havido algumas alterações e melhorias ao mesmo. Uma das melhorias foi a introdução de Timers ao algoritmo (Timer para o Prepare e Accept) de modo a que, caso haja uma falha no sistema, o protocolo reinicie e não espere internamente por respostas que não iram existir.

O pseudo-código definido no Algorithm 4 Paxos, mostra a forma como implementamos cada replica no Babel.

3.2.2 Multi Paxos. Através do material leccionado nas aulas, foi então possível estudar e implementar o algoritmo.

Este algoritmo encaixa-se na categoria do tipo de protocolos de *agreement*, como explicado anteriormente, que permite operações pedidas pela *State Machine*, cheguem a um consenso.

Durante a implementação do algoritmo, tivemos um cuidado especial com a eleição de um novo lider. Isto quer dizer que como o processo de eleição de um novo lider é perturbadora para o comportamento do sistema, após um lider antigo aperceber-se que está um novo processo a tentar ser o lider, o antigo tenta mandar uma operação do tipo *prepare* com um sequence number(na) maior, mesmo apesar de já ser tarde demais e o lider anterior ter sido descartado do sistema pelas outras instâncias. A implementação do Multi Paxos, difere do Paxos logo no metodo *uponProposeRequest*, que verifica se exite algum lider ou não. Caso não haja, o processo irá tentar tornar-se no novo lider, caso exista e no caso de ser o processo em questão, o mesmo procede imediatamente ao envio de *Accept Messages*.

No caso de outros processos receberem uma *Prepare Message*, caso não haja previamente lider no sistema, o mesmo é reconhecido como o processo que enviou o prepare. Caso o lider receba um prepare de outro processo, o mesmo invoca um *prepare* maior, mesmo sendo já demasiado tarde.

Algorithm 2 State Machine - Parte 2

```
1: function uponDecidedNotification( DecidedNotification, SourceProto )
2:     Trigger sendNotification(ExecuteNotification)
3:     executedOps ← executedOps + 1
4:     cumulativeHash ← appendOpToHash()
5:     if DecidedNotification.OpType == WRITE then
6:         data ← data ∪ DecidedNotification.Operation
7:     end if
8:     if opExecuting == null then
9:         Do Nothing
10:    end if
11:    if opExecuting == DecidedNotification.OpId then
12:        opExecuting ← null
13:        currentOp ← null
14:        if opsToBe.size() > 0 then
15:            Call uponOrderRequest(OrderRequest, protocolId)
16:        end if
17:    else
18:        Call uponOrderRequest(OrderRequest, protocolId)
19:    end if
20: end function
21: function uponInformMembership(InformMembership, host, SourceProto, ChannelId)
22:    membership ← InformMembership.membership
23:    for host ∈ membership do
24:        openConnection(host)
25:        Trigger send(JoinedNotification, membership, nextInstance)
26:    end for
27:    membership ← membership ∪ selfHost
28:    hasMembership = True
29:    if hasMembership == True and hasState == True then
30:        state ← ACTIVE
31:        Trigger send(JoinedNotification, membership, nextInstance)
32:        if opsToBe.size() > 0 then
33:            Call uponOrderRequest(OrderRequest, protocolId)
34:        end if
35:    end if
36: end function
```

Algorithm 3 State Machine - Parte 3

```
1: function uponOrderRequest( OrderRequest, SourceProto )
2:    if state == JOINING then
3:        opsToBe ← opsToBe ∪ OrderRequest.getOpId
4:        operationMap ← operationMap ∪ OrderRequest.getOperation()
5:    else
6:        if opExecuting == null then
7:            if agreement == "paxos" then
8:                send(ProposeRequest, PaxosProtocolId)
9:            else
10:               send(ProposeRequest, MultiPaxosProtocolId)
11:           end if
12:           opExecuting ← OrderRequest.OpId
13:           currentOp ← OrderRequest.Operation
14:           if SourceProto == selfProtocolId then
15:               opsToBe ← opsToBe \ OrderRequest.OpId
16:               operationMap ← operationMap \ OrderRequest.Operation
17:           end if
18:       else
19:           if SourceProto != selfProtocolId then
20:               opsToBe ← opsToBe ∪ OrderRequest.getOpId
21:               operationMap ← operationMap ∪ OrderRequest.getOperation()
22:           end if
23:       end if
24:   end if
25: end function
```

Os *timers* da implementação, tem como objetivo, garantir a propriedade de *liveness*, sendo que um deles é desencadeado pelo lider, de modo a que a cada x segundos seja enviado um *heartbeat*, pelo mesmo. O outro *timer* é desencadeado por todos os processos que esperem pelo *heartbeat* do lider.

De modo a ilustrar melhor o comportamento do algoritmo,de seguida apresenta-se o pseudocódigo do mesmo (Algoritmo1).

4 EXPERIMENTAL EVALUATION

4.1 Methodologies

Nesta avaliação utilizamos o Cluster do Departamento de Informática da Faculdade de Ciências e Tecnologias da Universidade Nova de Lisboa. Na fase de testes defrontamo-nos com inúmeros problemas que nos levaram a resultados falaciosos dos algoritmos em questão. Por esse motivo não nos foi possível apresentar resultados em tempo útil sobre os algoritmos implementados.

O nosso parecer sobre estes problemas vai ao encontro, talvez, da State Machine, que tendo sido a última componente a ser implementada, foi a que mais pecou em termos de veracidade na implementação tendo por isso levado a inconsistências em todo o sistema. Poderá também haver pequenos erros nos protocolos de Agreement que talvez não tenham sido resolvidos dados os erros da camada de cima.

Por esses motivos, neste capitulo iremos apresentar mais algumas informações sobre os algoritmos de Agreement de modo a complementar toda a informação até aqui falada de modo a que nos seja possível concluir com uma comparação teórica entre os dois tipos de mecanismos.

4.2 Results

4.2.1 Paxos. Por falta de resultados, não nos é possível falar sobre eles.

4.2.2 Multi-Paxos. No caso do Multi Paxos, não foi possível a aquisição de dados estatísticos, o problema que nós pensamos identificar que causou este problema, foi relacionado com os *timouts* de *Heart Beats* recebidos por instâncias não lider, levando assim às mesmas a uma tentativa de serem elas a tornarem-se as novas líder. Os resultados teóricos irão ser explicados na secção seguinte.

Algorithm 4 Paxos

```
1:  function Init
2:      prepareOkMessagesReceived ← 0
3:      acceptedMessages ← 0
4:  end function
5:  function uponProposeRequest( V )
6:      sequenceNumber ← getInstance()
7:      for member ∈ membership do
          Trigger send(PREPARE, sequenceNumber, member)
8:      end for
9:  end function

10: function uponPrepareMessage( receivedSequenceNumber, host )
11:     if receivedSequenceNumber > sequenceNumber then
12:         sequenceNumber ← receivedSequenceNumber Trigger send(PREPARE_OK, sequenceNumber host)
13:     end if
14: end function

15: function uponPrepareOkMessage( receivedSequenceNumber, host )
16:     if receivedSequenceNumber != sequenceNumber then return;
17:     end if
18:     prepareOkMessagesReceived ← prepareOkMessagesReceived+1
19:     if prepareOkMessagesReceived == getQuorumSize() then
20:         for member ∈ membership do Trigger send(ACCEPT, sequenceNumber, member)
21:         end for
22:     end if
23: end function

24: function uponAcceptMessage( receivedSequenceNumber, host )
25:     if receivedSequenceNumber >= sequenceNumber then
26:         sequenceNumber ← receivedSequenceNumber
27:         for member ∈ membership do
          . Trigger send(ACCEPT_OK, sequenceNumber, member)
28:         end for
29:     end if
30: end function

31: function uponAcceptOkMessage( receivedSequenceNumber, host )
32:     if receivedSequenceNumber > sequenceNumber then
33:         sequenceNumber ← receivedSequenceNumber
34:         acceptedMessages ← 1
35:     else
36:         if receivedSequenceNumber < sequenceNumber then //Nada a fazer
37:         else
38:             if ++acceptedMessages == getQuorumSize() then Trigger sendNotification(DECIDED, operation)
39:             end if
40:         end if
41:     end if
42: end function
```

Algorithm 5 Multi Paxos

```
1:  function uponProposeRequest(request)
2:      if lider == null then
3:          sequenceNumber ← request.getInstance()
4:          opId ← request.getOpId()
5:          operation ← request.getInstance()
6:          for member ∈ membership do
              Trigger send(PREPARE, sequenceNumber, member)
7:          end for
8:      else
9:          if lider == myself then
10:             for member ∈ membership do
                  Trigger send(ACCEPT, sequenceNumber, member)
11:             end for
12:         else, Trigger send(FORWARD, sequenceNumber, lider)
13:         end if
14:     end if
15: end function

16: function uponPrepareMessage(msg, host)
17:     receivedSequenceNumber ← request.getInstance()
18:     if myself == lider then
19:         sequenceNumber ← getNextSequenceNumber() Trigger uponProposeRequest(PROPOSE, sequenceNumber)
20:     else
21:         if receivedSequenceNumber > sequenceNumber then
22:             sequenceNumber ← receivedSequenceNumber
23:             if lider == null then Trigger send(PREPAREOK, host)
24:                 lider ← host
25:             elseTrigger send(INFORMMESSAGE, host)
26:             end if
27:         end if
28:     end if
29: end function

30: function uponPrepareOkMessage(msg, host)
31:     if msg.getSequenceNumber != sequenceNumber then
32:         return
33:     end if
          prepareOkMessagesReceived++
34:     if prepareOkMessagesReceived == getQuorumSize() then Trigger cancelTimer(prepareTimer)
35:         for member ∈ membership do
            Trigger send(ACCEPT, sequenceNumber, member)
36:         end for
37:         setPeriodicTimer(LIDERHEARTBEATTIMER, hbtime)
38:         if changed then
39:             for member ∈ membership do
                  Trigger send(REMOVELIDER, sequenceNumber, member)
40:             end for
41:             neigh ← neigh ∪ \ {(lider)})
42:             lider ← myself
43:             changed ← false
44:         end if
45:     end if
46: end function
```

Algorithm 6 Multi Paxos - Parte 2

```
1: function uponAcceptMessage(msg, host)
2:     receivedSequenceNumber ← msg.sequenceNumber()
3:     receivedOpId ← msg.getOpId()
4:     receivedOperation ← msg.getInstance()
5:     if receivedSequenceNumber >= sequenceNumber then
6:         sequenceNumber ← receivedSequenceNumbe
7:         opId ← receivedOpId
8:         operation ← receivedOperation
9:         for member ∈ membership do
       Trigger send(ACCEPTOK, sequenceNumber, member)
10:        end for
11:    end if
12: end function
13: function uponForwardingMessage(msg, host)
14:    receivedSequenceNumber ← msg.sequenceNumber()
15:    receivedOpId ← msg.getOpId()
16:    receivedOperation ← msg.getInstance()
17:    if receivedSequenceNumber >= sequenceNumber then
18:        sequenceNumber ← receivedSequenceNumbe
19:        opId ← receivedOpId
20:        operation ← receivedOperation
21:        for member ∈ membership do
       Trigger send(ACCEPTOK, sequenceNumber, member)
22:        end for
23:    end if
24: end function
25: function uponAcceptOKMessage(msg, host)
26:    receivedSequenceNumber ← msg.sequenceNumber()
27:    receivedOpId ← msg.getOpId()
28:    receivedOperation ← msg.getInstance()
29:    if receivedSequenceNumber > sequenceNumber then
30:        sequenceNumber ← receivedSequenceNumbe
31:        opId ← receivedOpId
32:        operation ← receivedOperation
33:    else
34:        if receivedSequenceNumber < sequenceNumber
   then, return
35:        end if
36:    end if
       acceptedMessages ← acceptedMessages +1
37:    if acceptedMessages == getQuorumSize() then
   cancelTimer(acceptTimer), Trigger DecideNotifica-
   tion(DECIDENOTIFICATION)
38:    end if
39: end function
40: function uponInformLiderMessage(msg, host)
       lider ← msg.getHost()
41:    for member ∈ membership do
42:        Trigger send(FORWARDREQUEST, sequenceNum-
   ber, opId, operation)
43:    end for
       Trigger setupPeriodicTimer(HearBeatWaitTimer, HB-
   TIMEOUT)
44: end function
```

Algorithm 7 Multi Paxos - P3

```
1: function uponHeartBeat(msg)
       cancelTimer(hbtimeout)
       Trigger setupPeriodicTimer(HearBeatWaitTimer, HB-
   TIMEOUT)
2: end function
3: function uponReelectionRequest(msg, host)
4:     neigh ← neigh ∪ \{(lider)})
5:     lider ← msg.getHost
6: end function
7: function uponSendHeartBeatTime(msg)
8:     for member ∈ membership do
9:         Trigger send(LIDERHEARTBEAT,member)
10:    end for
11: end function
12: function uponHeartBeatTimeout(msg)
13:    for member ∈ membership do
14:        Trigger send(PREPAREMESSAGE(getNextSequenceNumber()),
   member)
15:    end for
16: end function
```

4.3 Discussion

Como dito anteriormente, em prática não foi possível a aquisição de dados para ser feita uma analise estatística do comportamento dos servidores e dos clientes.

Em teoria, os resultados que deviam ter sido obtidos, deveriam mostrar um desempenho melhor do comportamento do sistema, quando utilizando o algoritmo de *Multi Paxos*, devido ao facto do mesmo algoritmo não oferecer tanta entropia no sistema, ao contrario do paxos. Isto deve-se pelo que no Multi Paxos, o líder é quem trata da gestão do pedidos efectuados pela State Machine e as outras instâncias do algoritmo, apenas remetem o pedido ao líder. Posteriormente, o facto de não serem necessários efectuar pedidos de *prepare*, a não ser numa reeleição, o desempenho do sistema deveria verificar uma melhoria na performance.

A propriedade *Liveness* que só por si o Paxos não conseguia garantir, é um problema que também influência o comportamento do Multi Paxos, devido a quando se detecta uma falha/crash num líder, é necessário haver uma reeleição do líder, posteriormente levando um líder activo a tentar permanecer no sistema. Este comportamento pode levar a haver condensação de prepares na rede se o algoritmo não estiver bem implementado.

5 CONCLUSIONS

De forma a concluir este projecto apresentamos agora a nossa opinião sobre o desenrolar do mesmo, que a nosso ver, poderia ter corrido melhor. A falta de resultados e falhas nas experiências localmente e no *Cluster* levou-nos a pensar que talvez fosse preciso mais algum tempo para levar a melhor porto este projecto. Apesar de não ter sido possivel a obtenção de dados, para efectuar uma analise de sensibilidade do comportamento do sistema no cluster, ainda assim foi possível desenhar e implementar uma solução de *State Machine Replication*, que em conjunto com um protocolo de *agreement*, quer seja o Paxos ou Multi-Paxos, possibilitasse a execução correcta e tolerante a falhas, de um sistema distribuído.

Em suma, e retirando os problemas que encontramos na fase final deste projecto, achamos que a passagem do conhecimento teórico foi conseguida tendo sido possível uma aplicação prática desses mesmos conhecimentos, tendo havido uma compreensão dos protocolos e arquitectura do sistema por parte dos membros do grupo.

REFERENCES

[1] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. *Paxos* made live: An engineering perspective.
Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing, PODC '07, page 398–407, New York, NY, USA, 2007. Association for Computing Machinery.

[2] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC '10, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery.

[3] Leslie Lamport. The part-time parliament. Paxos made simple.
ACM Trans. Comput. Syst., 16(2):133–169, May 1998.

[4] Leslie Lamport.
ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001), pages 51–58, December 2001.

[5] Robbert Van Renesse and Deniz Altinbuken. Paxos made moderately complex.
ACM Comput. Surv., 47(3), February 2015.

[6] Rubbert van Renesse.
State Machine Replication with Benign Failures, page 83–102. Association for Computing Machinery, New York, NY, USA, 2019.