Consenso Genérico em Sistemas Dinâmicos com Memória Compartilhada

Cátia Khouri[1,2], **Fabíola Greve**[1]

[1] D. de Ciência da Computação – U. Federal da Bahia (UFBA), Salvador, BA – Brasil

[2]DCET – U. Estadual do Sudoeste da Bahia, Vitória da Conquista, BA – Brasil

Abstract. *Consensus is a key service to design reliable applications on distributed dynamic systems. Unlike static systems, in such systems the set of participants is unknown and varies during the execution. This paper presents a general consensus for the shared memory model, subject to process crashes, with two innovative features: it does not require the knowledge of the cardinality of the system membership and it supports the use of both failure detectors and leader detector.*

Resumo. *O consenso é um serviço fundamental para o desenvolvimento de aplicações confiáveis sobre sistemas distribuídos dinâmicos. Diferentemente de sistemas estáticos, em tais sistemas o conjunto de participantes é desconhecido e varia ao longo da execução. Neste artigo, apresentamos um consenso genérico para o modelo de memória compartilhada, sujeito a falhas por parada, com duas características inovadoras: ele não pressupõe o conhecimento da cardinalidade do conjunto de processos em execução e suporta tanto o uso de detectores de falhas quanto de líder.*

1. Introdução

Sistemas distribuídos baseados em redes entre pares (P2P), computação em nuvem, redes de área de armazenamento (SANs), etc., são considerados dinâmicos no sentido de que os nós podem entrar e sair livremente, sem a necessidade de uma autoridade administrativa centralizada, de modo que o conjunto de participantes é auto-definido e variável. O dinamismo inerente a esses ambientes introduz uma imprevisibilidade adicional que deve ser levada em conta no projeto de algoritmos. Tais sistemas podem se beneficiar do consenso para atingir confiabilidade.

O consenso [Fischer et al. 1985] é uma abstração fundamental em sistemas distribuídos tolerantes a falhas onde não existe uma autoridade central para coordenar suas ações e que exige acordo entre os processos. Para que um conjunto de processos realize o consenso, cada processo propõe um valor e todos os processos corretos devem decidir por um único valor escolhido entre os propostos, a despeito de falhas no sistema. A maioria dos protocolos de consenso encontrados na literatura é voltada para sistemas em que a comunicação interprocessos se dá através de passagem de mensagens. Poucas são as propostas de algoritmos de consenso para sistemas em que os processos comunicam-se através de uma memória compartilhada, apesar de sua importância.

O modelo assíncrono de memória compartilhada condiz com as modernas arquiteturas multinúcleo, onde vários núcleos acessam uma mesma memória física. O estado da arte atual aponta para um cenário em que desde supercomputadores até simples sensores

serão multiprocessadores com memória compartilhada [Herlihy and Luchangco 2008]. Espera-se, já para o início da próxima década, o surgimento de máquinas *exascale*, isto é com capacidade de processamento da ordem de 10^{18} operações por segundo [TOP500 2014]. O gerenciamento de recursos nesses ambiente exige a disponibilidade de serviços de coordenação e acordo das mais variadas naturezas.

Outra classe importante de sistemas é a de serviços de armazenamento de dados de alta disponibilidade e tolerantes a falhas, tais como as *(i)* redes de área de armazenamento (SANs), *e.g.*, PASIS [Goodson et al. 2004]; *(ii)* sistemas de armazenamento P2P Bizantinos, *e.g.*, Rosebud [Rodrigues and Liskov 2004], OceanStore [Kubiatowicz et al. 2000]; e *(iii)* sistemas de passagem de mensagem nos quais servidores são modelados como componentes de armazenamento, *e.g.*, Fleet [Malkhi and Reiter 2000], Coca [Zhou et al. 2002].

Em uma SAN, um conjunto de discos conectados a uma rede (ou NAD – *Network Attached Disks*) é acessado diretamente pelos processos no sistema por requisições de escrita e leitura. Desse modo, um NAD pode ser visto como uma memória compartilhada com blocos de discos modelados como registradores compartilhados acessados concorrentemente por diversos processos. Como não é necessário qualquer conhecimento prévio sobre o conjunto de processos que participam no sistema, uma SAN é definida como um sistema dinâmico. Por isso, protocolos destinados a sistemas dinâmicos de memória compartilhada, podem ser mapeados para tais sistemas. Na realidade, as vantagens desta estrutura têm motivado o projeto de algoritmos de consenso baseados em disco [Aguilera et al. 2003, Chockler and Malkhi 2002, Gafni and Lamport 2003], que são capazes de fornecer serviços de armazenamento distribuídos confiáveis.

É bem sabido que não existe solução determinística para o consenso em sistemas assíncronos sujeitos a falhas, mesmo na ocorrência de uma única falha de processo [Fischer et al. 1985]. Detectores de falhas são uma alternativa elegante para contornar esta impossibilidade. Um detector de falhas da classe $\Diamond\mathcal{S}$ pode ser visto como um oráculo que fornece dicas sobre processos que falham [Chandra and Toueg 1996]. O oráculo líder Ω, fornece aos processos uma identidade de processo (supostamente) correto [Lamport 1998]. As classes de detectores $\Diamond\mathcal{S}$ e Ω são equivalentes no sentido de que são as mais fracas que permitem realizar o consenso em sistemas de passagem de mensagens [Chandra et al. 1996] ou de memória compartilhada [Delporte-Gallet et al. 2004].

Estudos do problema do consenso em sistemas de memória compartilhada, sujeitos a falhas, mostram que as condições suficientes e necessárias nem sempre são as mesmas de sistemas de passagem de mensagens [Guerraoui and Raynal 2007, Lo and Hadzilacos 1994, Neiger 1995]. Em particular, enquanto é requerida a maioria de processos corretos para resolver o consenso no modelo de passagem de mensagens, as soluções para ambientes de memória compartilhada podem ser *wait-free*, isto é, o protocolo funciona corretamente, mesmo que até $n - 1$ processos falhem.

Contribuição. Neste artigo, apresentamos um algoritmo genérico para o consenso em sistemas assíncronos, dinâmicos, de memória compartilhada, sujeitos a falhas de processos. O algoritmo é genérico no sentido de que pode ser instanciado com detectores da classe $\Diamond\mathcal{S}$ ou Ω. Alguns trabalhos fornecem soluções para sistemas estendidos ou com $\Diamond\mathcal{S}$ [Lo and Hadzilacos 1994] ou com Ω [Delporte-Gallet and Fauconnier 2009,

Neiger 1995], mas, até onde sabemos, não com ambos. O artigo [Khouri and Greve 2013] apresenta um tal consenso, mas, ele não suporta um dinamismo pleno, já que considera um conhecimento da cardinalidade n do conjunto de processos do sistema.

Recentemente, um estudo sobre condições de conectividade no grafo de conhecimento dos participantes do sistema foi realizado em [Khouri et al. 2013], a fim de resolver o consenso num ambiente dinâmico de memória compartilhada. O artigo apresenta um conjunto de protocolos que ampliam o conhecimento sobre os participantes do sistema através de uma prospecção no grafo de conhecimento, formado a partir de consultas a detectores de participação. O problema de consenso com participantes desconhecidos é, ao final, reduzido ao problema clássico do consenso com participantes conhecidos, mas que pertencem ao poço do grafo de conhecimento do sistema. Ou seja, o algoritmo de consenso pressupõe o conhecimento da cardinalidade n do conjunto de processos do poço.

Diferentemente de [Khouri and Greve 2013] e [Khouri et al. 2013], o algoritmo aqui proposto não assume o conhecimento de n, e então pode ser usado de maneira plena e direta em sistemas dinâmicos. Assim, ao nosso conhecimento, este artigo apresenta o primeiro protocolo de consenso genérico quanto ao oráculo para sistemas dinâmicos de memória compartilhada, sem que n seja conhecido. Considerando, portanto, a sincronia mínima necessária para resolver consenso, o algoritmo é ótimo. Também é ótimo no que diz respeito à resiliência, porque tolera qualquer número de falhas de processos ($n - 1$).

Na prática, um consenso genérico e modular é um arcabouço muito útil para construir sistemas dinâmicos de camadas superiores independentes do detector de falhas que está disponível. Assim, a implementação de consenso pode ser melhor adaptada às características particulares de cada ambiente; principalmente quando a implementação do detector serve a muitas aplicações. Desse modo, as aplicações existentes que já estão rodando sobre os detectores $\diamondsuit\mathcal{S}$ ou Ω podem ser portadas mais facilmente.

O restante do artigo está organizado como segue: a Seção 2 descreve o modelo do sistema. A Seção 3 apresenta o algoritmo genérico e a Seção 4 suas provas de corretude. A Seção 5 traz uma discussão sobre trabalhos relacionados e a Seção 6 conclui o artigo.

2. Modelo do Sistema

O sistema dinâmico é composto por um conjunto infinito de processos que se comunicam através de uma memória compartilhada e que podem falhar (*crash*), parando prematura ou deliberadamente (por exemplo, saindo arbitrariamente do sistema). Não fazemos suposições sobre a velocidade relativa dos processos para dar passos ou para realizar operações de acesso à memória compartilhada, isto é, o sistema é assíncrono. Para facilitar a apresentação, assumimos a existência de um relógio global que não é acessível pelos processos, cuja faixa de *ticks*, $\mathcal{T}$, é o conjunto dos números naturais.

2.1. Processos

Consideramos o *modelo de chegada finita* [Aguilera 2004], isto é, o sistema tem um número infinito de processos que podem entrar e sair em instantes arbitrários, mas em cada execução esse número é limitado, embora desconhecido. Existe um instante a partir do qual nenhum novo processo entra. Cada processo p_i localiza-se num nó distinto e possui uma identidade distinta, i, tal que o conjunto dessas identidades é totalmente ordenado. No decorrer do artigo, denotamos um processo por seu nome ou por sua identidade.

A despeito da entrada e saída de processos, é preciso contar com um conjunto de processos estáveis A noção de *estável* refere-se a um processo que entra no sistema e permanece ativo durante toda computação. Assim, admitimos a existência de pelo menos um processo estável no sistema. Note que para a conversão do algoritmo é suficiente que um processo permaneça ativo até que escreva uma decisão v em seu registrador e retorne v para a aplicação. Denotamos Π_{FA} tal conjunto de participantes do sistema, $\mid \Pi_{FA} \mid \leq 1$.

2.2. Memória Compartilhada

A memória compartilhada consiste de um arranjo $R[n]$ de registradores regulares do tipo 1-*escritor-n-leitores* ($1WnR$) que se comportam corretamente. Um registrador é um objeto compartilhado que aceita dois tipos de operação: leitura e escrita. O registrador $R[i]$ pertence ao processo p_i, que é o seu único escritor, mas pode ser acessado por qualquer p_j para leitura. Comportamento correto de um registrador significa que ele sempre pode executar uma leitura ou escrita e nunca corrompe o seu valor.

Uma operação invocada sobre um registrador compartilhado não é instantânea, ela dura um tempo desde a invocação até a resposta. Portanto, é possível que as operações invocadas por diferentes processos sobre um mesmo registrador se sobreponham no tempo. Um registrador *regular* é mais fraco que um atômico e comporta-se como segue [Lamport 1986]. Uma leitura que não sobrepõe a uma escrita, obtém o valor atual (último valor escrito). Uma leitura concorrente a uma ou mais escritas, pode retornar o valor antigo (escrito pela última operação de escrita não-concorrente à leitura), ou o valor sendo escrito por uma das escritas concorrentes. Além disso, se uma escrita sobrepõe a duas leituras sequenciais, a primeira leitura pode retornar o novo valor v'; e a segunda retornar o valor antigo v. Este fenômeno é chamado de inversão novo/antigo.

Propriedade 1 *Sejam os registradores regulares $R[i]$ e $R[j]$; e os processos p e q tais que: (1) p executa $write^p(R[i], v)$ e em seguida $read^p(R[j])$; (2) q executa $write^q(R[j], w)$ e em seguida $read^q(R[i])$. Então, se nenhuma outra operação é efetuada sobre $R[i]$ e $R[j]$, p obtém w na leitura em $R[j]$ ou q obtém v na leitura em $R[i]$.*

Prova. Suponha, por contradição, que p não obtém w e q não obtém v. Considere os seguintes instantes de início e término das operações executadas por p e q:

p: tw_b^p e tw_e^p: início e término da operação $write^p(R[i], v)$; tr_b^p e tr_e^p: início e término da operação $read^p(R[j])$. Portanto: $tw_b^p < tw_e^p < tr_b^p < tr_e^p \Rightarrow tw_e^p < tr_b^p$ (A).

q: tw_b^q e tw_e^q: início e término da operação $write^q(R[j], w)$; tr_b^q e tr_e^q: início e término da operação $read^q(R[i])$. Portanto: $tw_b^q < tw_e^q < tr_b^q < tr_e^q \Rightarrow tw_e^q < tr_b^q$ (B).

Da nossa hipótese, p não obtém w. Isso só pode acontecer, com registradores regulares, se $read^p(R[j])$ termina antes ou, possivelmente, é concorrente $write^q(R[j], w)$. Portanto, $tr_b^p \leq tw_e^q$ (C). Por outro lado, ainda pela hipótese, q não obtém v. Então, $tr_b^q \leq tw_e^p$ (D). De (C), (B), (D) e (A) temos: $tr_b^p \leq tw_e^q < tr_b^q \leq tw_e^p < tr_b^p$, uma contradição.

2.3. Consenso

No problema do consenso, cada processo p_i propõe um valor v_i e todo processo correto decide um mesmo valor v, entre os propostos. Mais precisamente, o consenso é definido pelas seguintes propriedades [Chandra and Toueg 1996, Fischer et al. 1985]:

Terminação: todo processo correto decide algum valor;
Validade: se um processo decide v, então v foi proposto por algum processo;
Acordo Uniforme: se um processo decide v, então todo processo que decide, decide v;

2.4. Detectores de Falhas

Para contornar a impossibilidade do consenso em sistemas assíncronos, uma abordagem válida é estender o sistema com detectores de falhas (ou líder), os quais proveem o sincronismo extra necessário [Chandra and Toueg 1996]. Um detector de falhas ($\mathcal{D}$) funciona como um oráculo distribuído que fornece dicas sobre processos faltosos (ou corretos).

Detector de Falhas $\diamondsuit\mathcal{S}$. Os detectores de falhas podem ser classificados de acordo com as propriedades de completude e acurácia que exibem. Neste trabalho, o nosso interesse está sobre a classe de detectores *forte após um tempo* (*eventually strong*, ou $\diamondsuit\mathcal{S}$). Quando invocado por um processo, ele fornece uma lista de processos suspeitos. Num contexto de sistema dinâmico, um detector $\diamondsuit\mathcal{S}$ satisfaz às seguintes propriedades:

Completude forte (*Strong completeness*). Após um tempo, todo processo faltoso será considerado permanentemente suspeito por todo processo estável.

Acurácia fraca após um tempo (*Eventual weak accuracy*). Existe um instante após o qual algum processo estável jamais será considerado suspeito por qualquer processo estável.

Detector de Líder Ω . O detector de líder após um tempo, conhecido como Ω, também é um tipo de oráculo distribuído que provê um processo p_i com uma função local $leader()$ que retorna a identidade de um processo considerado correto. Considerando um ambiente dinâmico, um detector Ω satisfaz à seguinte propriedade:

Liderança após um tempo (*eventual leadership*): existe um instante após o qual qualquer invocação de $leader()$ por qualquer processo estável p_i, retorna a identidade do mesmo processo estável p_l.

Note que os instantes de estabilidade garantidos pelas propriedades de Ω e $\diamondsuit\mathcal{S}$ não são conhecidos pelos processos.

3. Algoritmo Genérico

Nesta seção apresentamos o algoritmo executado por todo processo p_i. A função CONSENSUS $(v_i, \mathcal{D})$ consiste de um laço infinito que recebe o valor inicial de p_i e pode ser instanciada com um oráculo da classe Ω ou da classe $\diamondsuit\mathcal{S}$. O valor inicial de p_i então, passa a ser sua "estimativa" para uma possível decisão. O algoritmo funciona em rodadas, mas nem todos os processos avançam nas rodadas. O número da rodada correntemente executada por p_i, r_i, é atualizado seguindo uma sequência estritamente crescente.

Em cada iteração do laço, p_i define um processo como proponente ($proposer_i$), de acordo com o oráculo instanciado. Processos proponentes invocam PROPOSITION $(v_i, \&r_i)$ para tentar impor uma decisão, enquanto que não proponentes limitam-se a aguardar a decisão do proponente para então copiá-la. Se $\mathcal{D} = \Omega$, o proponente é o líder retornado pelo oráculo. Se $\mathcal{D} = \diamondsuit\mathcal{S}$, a escolha do proponente segue o paradigma do coordenador rotativo adaptado para um ambiente dinâmico. Na primeira iteração, o proponente é o processo de identidade 1 ($proposer_i = p_1$) e a cada nova iteração essa identidade é incrementada de 1. Quando a identidade do proponente alcança a maior identidade de processo que já se ligou ao sistema, a contagem recomeça de 1. Assim, enquanto um valor não é decidido, todo processo tem a chance de ser proponente.

Como os detectores Ω e $\diamondsuit\mathcal{S}$ não são confiáveis, pode acontecer, por exemplo, de Ω prover líderes distintos para processos distintos, ou $\diamondsuit\mathcal{S}$ suspeitar erroneamente de pro-

cessos corretos. Em ambos os casos, o resultado é a coexistência de vários proponentes tentando impor suas decisões. Então, o algoritmo é indulgente para com o detector, isto é, garante segurança (*safety*) durante os períodos de instabilidade, atingindo vivacidade (*liveness*) quando o sistema se estabiliza [Guerraoui and Raynal 2003]. Assim, quando um proponente p_i, na rodada r_i, percebe a presença de outro proponente p_j, na rodada $r_j \geq r_i$, desiste de tentar decidir. Então p_i retorna para CONSENSUS e volta a definir um proponente. Quando um não proponente percebe que o proponente de quem ele esperava uma decisão abandonou a rodada, também parte para uma nova iteração definindo um novo proponente. Enquanto novos proponentes são continuamente definidos, aqueles anteriores (com números de rodada menores), continuamente desistem de decidir, e nesse período de anarquia, é possível que nenhuma decisão seja tomada. Felizmente, em algum instante depois que as propriedades dos detectores passarem a ser satisfeitas, haverá um único proponente que decidirá um valor e o escreverá em seu registrador, possibilitando a que os demais processos tomem a mesma decisão e, então, o algoritmo converge.

Cada registrador é composto dos seguintes campos:

$R[i].rd$: inteiro que indica a rodada corrente executada por p_i. Inicializado com 0.
$R[i].vl$: valor que pode representar uma *estimativa*, *proposta* ou *decisão* de p_i. Inicializado com $\perp$, que denota um valor que não pode ser proposto por qualquer processo.
$R[i].tg$: rótulo que discrimina o valor em $R[i].vl$: est, pro ou dec. Inicializado com $\perp$.

As operações executadas sobre os registradores são:

$read(R[i], aux)$: lê o registrador $R[i]$, retornando seu valor para a variável local aux.
$write(R[i], aux)$: escreve o valor da variável local aux no registrador $R[i]$.

Além dos registradores compartilhados, cada p_i possui variáveis locais:

- r_i – número da rodada corrente de p_i;
- $proposer_i$ – identidade do proponente corrente;
- aux – arranjo de registros (usado como variável auxiliar para ler R);
- max_rd – número da maior rodada em R;
- e_i – estimativa de decisão de p_i;
- $decision_i$ – valor decidido;
- $prop$ – registro (usado como variável auxiliar para ler um registrador compartilhado).

3.1. Descrição do Algoritmo

O protocolo é composto de duas funções: CONSENSUS $(v_i, \mathcal{D})$, que é executada por todo p_i, onde v_i é o valor inicial de p_i e $\mathcal{D}$ é o detector utilizado; e PROPOSITION $(v_i, \&r_i)$, que só é executada por p_i quando ele se considera proponente (i.e., $proposer_i = i$), onde r_i é a rodada corrente de p_i. No início da execução de CONSENSUS, p_i define $proposer_i$ conforme o detector (linhas 3-4 ou 5-9) e então segue um dos caminhos:

(1) Se $proposer_i = i$, p_i invoca PROPOSITION $(v_i, \&r_i)$ (linhas 10-12). Então p_i ajusta sua estimativa corrente, e_i, para seu valor inicial, v_i, e incrementa o número da rodada (linha 27). É nesta função que uma decisão ocorre primeiro. A informação constante no registrador de um proponente bem sucedido, começa como uma *estimativa* que vai amadurecendo, passa pelo estágio de *proposta* até alcançar o estado de *decisão*. Isso acontece em duas fases quase idênticas – linhas 28-42 e 43-54.

Na fase 1, p_i escreve em $R[i]$ seus dados correntes, r_i e e_i, e o rótulo "est", que identifica o estágio de maturidade da informação (linha 28). Em seguida, p_i lê o arranjo de

```
1: r_i ← 0; proposer_i ← 0; aux = ∅;
2: loop
3:     if (D ∈ Ω) then
4:         proposer_i ← leader();
5:     else if (D ∈ ◇S) then
6:         if (proposer_i ≤| aux |) then
7:             proposer_i ← proposer_i + 1;
8:         else
9:             proposer_i ← 1;
10:    if (proposer_i = i) then
11:        decision =PROPOSITION(v_i, D);
12:        if (decision ≠ ⊥) then return decision;
13:    else
14:        if (D ∈ Ω) then
15:            repeat;
16:                read (R[proposer_i], prop);
17:            until ((prop.tg = dec) ∨ (proposer_i ≠ leader()));
18:        else if (D ∈ ◇S) then
19:            repeat
20:                read (R[proposer_i], prop);
21:            until ((prop.tg = dec) ∨ (proposer_i ∈ suspected_i));
22:            if ( (prop.tg ≠ dec) ∨ ( (prop.tg =dec) ∧ (prop.vl = ⊥) ) ) then
23:                read (R, aux);
24:        if ( (prop.tg = dec) ∧ (prop.vl ≠ ⊥) ) then
25:            write (R[i], prop);
26:            return prop.vl;
```

Figura 1. Algorithm CONSENSUS $(v_i, \mathcal{D})$

registradores R (linha 29). Se obtém alguma decisão, adota-a como a sua e retorna para CONSENSUS (linhas 30-32). Se por outro lado, percebe que há algum outro proponente em uma rodada maior ou igual à sua, p_i desiste de tentar impor uma decisão, escreve um valor inválido ($\perp$) e o rótulo "dec" em $R[i]$ e retorna $\perp$ (linhas 34-39). Outra verificação como esta é feita na fase 2. Isso impede que dois (ou mais) proponentes concorrentes decidam valores distintos. Se, ainda, p_i obtém alguma proposta em R, altera sua estimativa para este valor (linhas 41-42) e a escreve em $R[i]$, atualizando seu estado para "pro".

Na fase 2, p_i lê os registradores compartilhados, verifica a existência de proponentes mais adiantados e se for o caso, desiste da rodada (linhas 43-49). Se no entanto ele persiste, verifica se há alguma decisão em R e, em caso positivo, altera sua estimativa para este valor (linhas 51-52). Por fim, p_i dissemina sua decisão escrevendo-a em $R[i]$, com o rótulo apropriado, "dec", retornando para CONSENSUS (linhas 53-54). Uma vez que um valor $v \neq \perp$ é retornado, p_i retorna v para a aplicação (linhas 11-12). Nos casos em que PROPOSITION retorna $\perp$, p_i segue para a próxima iteração (linha 2).

(2) Se $proposer_i \neq i$, p_i permanece em um **repeat-until** esperando obter uma decisão do proponente (linhas 14-17 ou 18-21). No entanto, se p_i suspeita do proponente, abandona

```
27: e_i ← v_i; r_i ← r_i + 1;
28: write (R[i], ⟨r_i, e_i, est⟩)
29: read (R[1..], aux[1..]);
30: if (∃j : aux[j].tg = dec) then
31:     write(R[i], ⟨−, aux[j].vl, dec⟩);
32:     return aux[j].vl;
33: else
34:     if (∃ j : aux[j].rd ≥ r_i) then
35:         max_rd ← j : ∀j, k : aux[j].rd ≥ aux[k].rd;
36:         if (max_rd > r_i) then
37:             r_i ← max_rd;
38:             write(R[i], ⟨−, ⊥, dec ⟩);
39:             return ⊥;
40:     else
41:         if (∃j : (aux[j].tg = pro) then
42:             e_i ← (aux[j].vl: ∀j, k:aux[j].tg = aux[k].tg=pro, aux[j].rd ≥ aux[k].rd);
43:         write (R[i], ⟨r_i, e_i, pro⟩);
44:         read (R[1..ns], aux[1..ns]);
45:         if (∃ j : aux[j].rd > r_i) then
46:             max_rd ← j : ∀j, k : aux[j].rd ≥ aux[k].rd;
47:             r_i ← max_rd;
48:             write(R[i], ⟨−, ⊥, dec ⟩);
49:             return ⊥;
50:         else
51:             if (∃j : aux[j].tg = dec) then
52:                 e_i ← (aux[j].vl: ∀j, k:aux[j].tg = aux[k].tg=dec, aux[j].rd ≥ aux[k].rd);
53:             write (R[i], ⟨r_i, e_i, dec⟩);
54:             return e_i;
```

Figura 2. Algorithm PROPOSITION $(v_i, \&r_i)$

a espera e começa nova iteração. Se $\mathcal{D} = \Omega$, a suspeita se dá quando Ω retorna uma identidade diferente do proponente corrente (linha 17). Se $\mathcal{D} = \diamondsuit\mathcal{S}$, o proponente aparece na lista de suspeitos (linha 21). Note que quando $proposer_i$ abandona a rodada, escreve um valor inválido com rótulo "dec" (linhas 38, 48), o que também libera p_i da espera. No caso do detector $\diamondsuit\mathcal{S}$, antes de p_i iniciar uma nova iteração, lê os registradores R a fim de obter informações necessárias à definição do novo proponente, a saber, o tamanho corrente do conjunto de participantes no sistema (linhas 22-23). Por fim, se p_i obtém uma decisão $v \neq \perp$, decide o mesmo valor, o qual retorna para a aplicação (linhas 24-26).

4. Prova de Corretude

Esta seção apresenta um esquema da prova de que o Algoritmo 1 satisfaz às propriedades do Consenso. Um processo p_i invoca CONSENSUS$(v_i, \mathcal{D})$, com seu valor inicial v_i como argumento, que passa a ser sua estimativa (linha 1) para uma possível decisão (validade). Então p_i entra em um laço (linha 2) do qual só sai após decidir um valor (a menos que falhe ou deixe o sistema). Quando p_i é proponente, tenta decidir um valor que será copiado por

todo p_j tal que $proposer_j = i$. Mas, como dito na Seção 3, mais de um processo pode considerar-se proponente ao mesmo tempo. Nas provas a seguir, atenção especial será dada a este caso para garantir que uma mesma decisão seja tomada por todos (acordo) e que os processos estáveis em algum instante cheguem a uma decisão (terminação).

Notação. Consideramos que "um processo p_i propõe um valor v" quando p_i escreve v em seu registrador junto com tag: pro, isto é, executa a operação **write**$(R[i], \langle -v, \text{pro}\rangle)$. Da mesma forma, consideramos que "p_i decide um valor v" quando executa a operação **write**$(R[i], \langle -, v, \text{dec}\rangle)$ e retorna o valor v. Denotamos SM-$\mathcal{AS}[\Pi_{\mathsf{FA}}, f, \mathcal{D}]$ um sistema de memória compartilhada, assíncrono, dinâmico (chegada finita – *finite arrival*), em que até f processos podem falhar por parada, estendido com um detector Ω ou $\diamondsuit\mathcal{S}$.

Lema 1 (Validade) *Em um sistema* SM-$\mathcal{AS}[\Pi_{\mathsf{FA}}, f, \mathcal{D}]$*, se algum processo* $p_i \in \Pi_{\mathsf{FA}}$ *invoca* CONSENSUS$(-, \mathcal{D})$ *e decide* v*, então* v *é o valor inicial de algum processo.*

Prova. Um processo p_i pode decidir um valor executando as linhas 25, 31 ou 53. Nas linhas 25 e 31, p_i decide um valor v que já foi decidido, obtido por ele em algum registrador (linhas 16 ou 20, 24-25; e 29-31; respectivamente). Então, é suficiente ver o caso em que p_i decide na linha 53. Nesse caso, o valor decidido é o que está armazenado em e_i.

Todo $R[j].vl$ é inicializado com $\perp$. Na linha 27, p_i atribui a e_i seu valor inicial v_i. Assim, na primeira vez que $p_i, \forall i$, escreve em $R[i].vl$, escreve seu valor inicial (linha 28). Nas demais operações de escrita, o valor escrito por p_i é seu valor inicial (atribuído a e_i na linha 27) ou o valor obtido em algum $R[j]$ e atribuído a e_i (linhas 29, 41-43 ou 44, 51-53). Portanto, todo valor $v \neq \perp$ em $R[j].vl, \forall j$, é o valor inicial de algum processo. Consequentemente, se p_i decide v, v é o valor inicial de algum processo. $_{Lema1}\square$

Lema 2 (Acordo) *Em um sistema* SM-$\mathcal{AS}[\Pi_{\mathsf{FA}}, f, \mathcal{D}]$*, se algum processo* $p_i \in \Pi_{\mathsf{FA}}$ *invoca* CONSENSUS$(-, \mathcal{D})$ *e decide* v*, e outro processo* $p_j \in \Pi_{\mathsf{FA}}$ *invoca* CONSENSUS$(-, \mathcal{D})$ *e decide* u*, então* $u = v$*.*

Prova. Um processo p_i decide um valor v ao executar a linha 25, 31 ou 53. Se ele decide na linha 25 ou 31, v é um valor que foi decidido por algum p_j e obtido por p_i, respectivamente, na linha 16 ou 20; ou na linha 29, de modo que o Lema é satisfeito. Considere, então, que p_i decide na linha 53. Suponha, sem perda de generalidade, que dois processos, p_i e p_j, são os únicos proponentes em um intervalo de tempo I, em que p_i, executando a rodada r_i, decide v e é o primeiro processo a decidir; e p_j, executando a rodada r_j, decide u ou abandona a rodada sem decidir. Vamos analisar os seguintes casos:

Caso 1: $r_j > r_i$. Considere as operações de escrita e leitura executadas por p_i nas linhas 43 e 44, respectivamente; e as operações de escrita e leitura executadas por p_j nas linhas 28 e 29, respectivamente. Pela Propriedade 1, p_i obtém $R[j] = \langle r_j, -, -\rangle$ na linha 44, ou p_j obtém $R[i] = \langle r_i, e_i, \text{pro}\rangle$ (ou $\langle r_i, e_i, \text{dec}\rangle$, se p_i já tiver executado a linha 53) na linha 29. Se p_i escreve $\langle -, v, \text{dec}\rangle$, na linha 53, é porque não obteve $R[j] = \langle r_j, -, -\rangle$ na linha 44, ou teria desistido da rodada (linhas 45-49) sem atingir a linha 53. Portanto, quando p_j executa a linha 29, obtém (a) $R[i] = \langle r_i, v, \text{pro}\rangle$; ou (b) $R[i] = \langle r_i, v, \text{dec}\rangle$.

(a) Como, da nossa hipótese, p_i e p_j são os únicos proponentes em I, se p_j obtém $R[i] = \langle r_i, v, \text{pro}\rangle$, avalia os predicados das linhas 30 e 34 como *falso* e o da linha 41 como *verdadeiro*. Então atribui v a e_j na linha 42. Em sua segunda leitura sobre R (linha 44), p_j obtém, novamente, $R[i] = \langle r_i, v, \text{pro}\rangle$, ou $R[i] = \langle r_i, v, \text{dec}\rangle$. Em qualquer

dos casos, p_j avalia o predicado da linha 45 como *falso* e, independente da avaliação do predicado da linha 51, executa a operação **write**$(R[j], \langle r_j, e_j, \text{dec}\rangle)$ com $e_j = v$ e o Lema é mantido.

(b) Se p_j obtém $R[i] = \langle r_i, v, \text{dec}\rangle$, na linha 29, avalia o predicado da linha 30 como *verdadeiro* e executa **write**$(R[j], \langle r_j, v, \text{dec}\rangle)$, de modo que o Lema é satisfeito.

Caso 2: $r_j = r_i$. Considere as operações de escrita e leitura executadas nas linhas 28 e 29, respectivamente, por p_i e p_j. Pela Propriedade 1, p_i obtém $R[j] = \langle r_j, -, -\rangle$, ou p_j obtém $R[i] = \langle r_i, -, -\rangle$. Se p_i decide v na linha 53, é porque não obteve $R[j] = \langle r_j, -, -\rangle$ na leitura da linha 29, caso contrário, p_i teria desistido da rodada, contrariando a nossa hipótese. Então, p_j obtém $r_i = r_j$ (linha 29) e desiste da rodada escrevendo $\langle -, \perp, dec\rangle$, de modo que o Lema á satisfeito.

Caso 3: $r_j < r_i$. Considere as operações de escrita e leitura das linhas 43 e 44, executadas por p_j na rodada r_j, bem como as operações de escrita e leitura das linhas 28 e 29, executadas por p_i na rodada r_i, respectivamente. Pela Propriedade 1, p_i obtém $R[j] = \langle r_j, u, \text{pro}\rangle$ (ou valor mais atualizado); ou p_j obtém $R[i] = \langle r_i, v, \text{est}\rangle$.

(a) Se p_j obtém $R[i] = \langle r_i, v, \text{est}\rangle$, avalia o predicado da linha 45 como verdadeiro e desiste da rodada, de modo que o Lema se mantém.

(b) Se p_i obtém $R[j] = \langle r_j, e_j, \text{pro}\rangle$ na linha 29, uma vez que p_i e p_j são os únicos proponentes, e p_i decide primeiro, p_i avalia os predicados das linhas 30 e 34: *falso*. Seja $e_j = v$. Na linha 41 p_i avalia o predicado: *verdadeiro*, de modo que atribui v a e_i na linha 42. Na linha 44, p_i obtém, novamente, $R[j] = \langle r_j, u, \text{pro}\rangle$. Mais uma vez, como p_i e p_j são os únicos proponentes, p_i avalia os predicados das linhas 45 e 51: *falso* e decide $e_i = v$ na linha 53. Então, se p_j não obtém $R[i] = \langle r_i, v, \text{est}\rangle$ na linha 44, avalia o predicado das linhas 45 e 51: *falso* e decide v na linha 53, satisfazendo o Lema. $_{Lema2}\square$

Lema 3 *Em um sistema $\mathcal{AS}[\Pi, f, \mathcal{D}]$, $\mathcal{D}$ = ($\diamondsuit\mathcal{S}$ ou Ω), pelo menos um processo estável que invoca PROPOSITION$(-, r_i)$ decide um valor v.*

Prova. Conforme explicado na Seção 3, devido à não confiabilidade dos detectores de falhas, é possível que durante um período, suspeições equivocadas permaneçam provocando o abandono da rodada por parte de proponentes sem que nenhuma decisão seja tomada. Mas, considerando que pelo menos um processo permanece ativo para sempre, e conforme o oráculo instanciado, temos que:

(1) Se $\mathcal{D} = \Omega$, a propriedade *liderança após um tempo* garante que a partir de um instante t, $leader()$ retorna sempre a mesma identidade de um processo estável p_s para todo processo estável no sistema. Então, ao cabo de um tempo, $proposer_i = s, \forall i$.

(2) Se $\mathcal{D} = \diamondsuit\mathcal{S}$, a propriedade *acurácia fraca após um tempo* garante que a partir de um instante t, algum processo estável p_s jamais será considerado suspeito por qualquer processo estável. Portanto, após t, seguindo o rodízio na definição do proponente, p_s acabará sendo considerado $proposer_i, \forall i$.

Uma vez que, se um processo abandona uma rodada, pula para o número de rodada mais alto em R (linha 37 ou 47) e a cada invocação de PROPOSITION, o número de rodada é incrementado de 1 (linha 27), após um tempo, p_s terá o número de rodada mais alto em R. Então, como p_s não é mais considerado suspeito, acaba decidindo v na linha 53. $_{Lema3}\square$

Lema 4 *Em um sistema $\mathcal{AS}[\Pi, f, \mathcal{D}]$, $\mathcal{D}$ = ($\diamondsuit\mathcal{S}$ ou Ω), nenhum processo estável que invoca CONSENSUS$(-, \mathcal{D})$ permanece bloqueado indefinidamente.*

Prova. O único trecho do algoritmo em que um processo p_i pode ficar bloqueado é no **repeat-until** das linhas 15-17 ou 19-21. Em qualquer caso, p_i fica aguardando que $proposer_i$ decida um valor. Para garantir que p_i não fica indefinidamente bloqueado, temos que mostrar que uma das condições é satisfeita: *Condição 1:* $R[proposer_i].tg = \text{dec}$; *Condição 2:* $proposer_i \neq leader()$ (se $\mathcal{D} = \Omega$); *Condição 3:* $proposer_i \in suspected_i$ (se $\mathcal{D} = \diamondsuit\mathcal{S}$). Todas as condições dependem do comportamento de $proposer_i$ e dos oráculos. Temos então que: (1) Se $proposer_i$ decide v, escreve sua decisão em seu registrador (linhas 31 ou 53) e a *Condição 1* é satisfeita. (2) Se $proposer_i$ falha ou deixa o sistema, então, (2.1) se $\mathcal{D} = \Omega$, devido à propriedade *liderança após um tempo*, em algum instante t, $leader()$ retorna a identidade de um processo estável para todo processo estável no sistema e a *Condição 2* é satisfeita. (2.2) Se $\mathcal{D} = \diamondsuit\mathcal{S}$, a propriedade *completude forte* garante que em algum momento $proposer_i \in suspected_i$ e a *Condição 3* é satisfeita. (3) Se $proposer_i$ abandona a rodada sem decidir um valor, antes de retornar de PROPOSITION, $proposer_i$ escreve $\langle -, \perp, \text{dec}\rangle$ em seu registrador, e então a *Condição 1* é satisfeita. (4) Se $proposer_i$ não se considera proponente, não invoca a função PROPOSITION $(v_i, \&r_i)$. Mas, pelo Lema 3, pelo menos um processo estável decide um valor v, e sua prova é baseada no fato de que após um instante t, todo processo p_i considerará o mesmo p_s como proponente, o qual decide v. Então, os demais processos estáveis, ao cabo de um tempo, obtém $R[s]$ tal que $R[s].tg =$ "dec", e a *Condição 1* é satisfeita. $Lema4\square$

Lema 5 (Terminação) *Em um sistema $\mathcal{AS}[\Pi, f, \mathcal{D}]$, $\mathcal{D}$ = ($\diamondsuit\mathcal{S}$ ou Ω), todo processo estável que invoca CONSENSUS$(-, \mathcal{D})$ decide um valor v.*

Prova. Pelo Lema 4, nenhum processo fica bloqueado executando CONSENSUS. Pelo Lema 3, ao cabo de um tempo, pelo menos um processo estável p_s decide um valor e é tal que $proposer_i = s, \forall i$ (veja a prova do Lema 3), decide v. Então, após este instante, todo processo estável obtém o valor decidido (linhas 16 ou 20) e decide o mesmo valor (linhas 24-26). $Lema5\square$

Teorema 1 *Em um sistema $\mathcal{AS}[\Pi, f, \mathcal{D}]$, $\mathcal{D}$ = ($\diamondsuit\mathcal{S}$ ou Ω), o Algoritmo 1 (com a ajuda do Algoritmo 2) satisfaz as propriedades do Consenso (definidas em 2.3).*

Prova. Segue diretamente da lemmata 1, 2 e 5. $Teorema1\square$

Teorema 2 *O Algoritmo 1 (juntamente com o Algoritmo 2) é wait-free.*

Prova. O Algoritmo 1 é correto, a despeito de $(n-1)$ falhas (Teorema 1). Considere uma execução em que exatamente $(n-1)$ processos falham. Seja p_i o único processo estável nesta execução. Devido ao rodízio na definição do coordenador e à propriedade de completude forte de $\diamondsuit\mathcal{S}$; ou à propriedade de liderança eventual de Ω, em um instante t, $proposer_i = i$. Ao executar PROPOSITION, p_i não encontra qualquer rodada superior à sua em R (linhas 34–39 e 45–49). Então, ele consegue atingir a linha 53, quando decide um valor.

Considere agora uma execução na qual menos de $(n-1)$ processos falham. Pelo Lema 3, ao menos um processo estável p_s decide um valor. Na prova desse Lema, vemos que, devido às propriedades dos detectores e ao rodízio na definição do proponente (para $\mathcal{D} = \diamondsuit\mathcal{S}$), a partir de um instante t, se nenhuma decisão é tomada antes, $proposer_i = s, \forall i$. Pelo Lema 4, nenhum p_i fica indefinidamente bloqueado. Assim, p_s escreve sua decisão em $R[s]$ e todo processo estável a obtém (linhas 16 ou 20), decidindo em seguida pelo mesmo valor (linhas 24-26). $Teorema2\square$

5. Discussão e Trabalhos Relacionados

O consenso tem sido amplamente estudado e vários algoritmos propostos, a maioria deles para sistemas de passagem de mensagens [Chandra and Toueg 1996, Lamport 1998, Guerraoui and Raynal 2003]. Para o modelo de memória compartilhada, foram identificados alguns artigos [Guerraoui and Raynal 2007, Delporte-Gallet et al. 2004, Lo and Hadzilacos 1994], além do nosso.

[Lo and Hadzilacos 1994] propõem um algoritmo de consenso para um conjunto estático de n de participantes, estendido com um detector $\diamondsuit\mathcal{S}$. Eles provam que são necessários pelo menos n registradores *atômicos* $1WnR$ para construir algoritmos de consenso *wait-free*, usando um detector da classe *forte*. O nosso algoritmo funciona corretamente com n registradores $1WnR$ regulares (mais fracos), utilizando detectores de falhas mais fracos ($\diamondsuit\mathcal{S}$ ou Ω), onde n é o número de processos que se ligou ao sistema até o instante da decisão. [Delporte-Gallet et al. 2004] propõem um algoritmo para um conjunto de participantes ilimitado, mas utilizam um número ilimitado de registradores bem mais complexos, isto é, registradores atômicos $MWMR$.

[Guerraoui and Raynal 2003] identificam a estrutura de informação de algoritmos de consenso indulgentes. Eles apresentam um algoritmo de consenso genérico com relação ao oráculo e mantém a mesma complexidade de acordo com alguns critérios. Diferentemente do nosso, o modelo considerado é o de passagem de mensagens e o número máximo de falhas é $f < n/2$. Em [Guerraoui and Raynal 2007], eles apresentam um arcabouço que unifica uma família de algoritmos de consenso com base no detector Ω que pode ser ajustada para diferentes modelos de comunicação, mas em qualquer um dos modelos só consideram Ω.

Em [Khouri et al. 2013] é apresentado um estudo do problema do consenso num ambiente dinâmico, cuja abordagem consiste em reduzir o problema do consenso com participantes desconhecidos ao problema clássico do consenso com conjunto de participantes conhecido. [Khouri and Greve 2013] propõem um algoritmo de consenso para memória compartilhada no qual o número de processos no sistema é conhecido e fixo.

Diferente dos trabalhos anteriores, o algoritmo aqui proposto não pressupõe conhecimento de n podendo ser usado de maneira plena e direta no ambiente dinâmico. Essa independência de n é fundamental, tanto teórica (porque abre nova linha de protocolos de consenso em sistemas dinâmicos – diferente da abordagem com detectores de participantes), quanto prática (porque pode ser usado sob qualquer camada de memória compartilhada dinâmica).

Complexidade O algoritmo usa n registradores, um para cada processo que entra no sistema. Embora não tenhamos identificado trabalhos que estabeleçam o número mínimo de registradores requeridos para resolver o consenso num ambiente dinâmico de memória compartilhada, o limite em [Lo and Hadzilacos 1994] para sistemas estáticos é um forte indício de que nosso algoritmo apresenta complexidade satisfatória. Entretanto, uma vez que o número de rodadas não é limitado, o tamanho dos registradores é ilimitado também.

Quanto à complexidade de passo, o nosso algoritmo apresenta o mesmo custo do proposto em [Guerraoui and Raynal 2007] que usa um detector Ω. Sabe-se que em sistemas assíncronos, um algoritmo de consenso pode ter uma execução ilimitada. Mas um cenário comum na prática, é quando durante alguns períodos, a comunicação ocorre de

maneira sincronizada. Nesses períodos, detectores não confiáveis podem ser bem precisos. [Keidar and Rajsbaum 2001] chamam de *bem comportada* uma execução livre de falhas na qual desde o início o sistema comporta-se de maneira síncrona e o oráculo é preciso, isto é, não comete erros. Considerando tais execuções, para decidir um valor, um proponente p_i consegue chegar a uma decisão em sua primeira rodada, executando três operações de escrita (em $R[i]$ – linhas 28, 43 e 53) e duas operações de leitura sobre o arranjo de registradores para decidir (linhas 29 e 44). Os demais processos obtém a decisão no registrador do proponente (linha 16 ou 20), o que pode ocorrer após uma única leitura.

6. Conclusão

Neste artigo, apresentamos um algoritmo genérico para resolver consenso tolerante a falhas em um sistema assíncrono de memória compartilhada, estendido com $\diamondsuit\mathcal{S}$ ou Ω. Mostramos que é possível construir um algoritmo genérico baseado em oráculo para o modelo de memória compartilhada, em que a composição do sistema é desconhecida, assim como a sua cardinalidade. Mais precisamente, o algoritmo proposto não assume o conhecimento de n e então pode ser usado de maneira plena e direta em sistemas dinâmicos. Ao nosso conhecimento, apresentamos o primeiro protocolo de consenso genérico quanto ao oráculo para sistemas dinâmicos de memória compartilhada, sem que n seja conhecido. Considerando, portanto, a sincronia mínima necessária para resolver consenso, o algoritmo é ótimo. Também é ótimo no que diz respeito à resiliência, porque tolera qualquer número de falhas de processos ($n - 1$).

Nosso resultado contribui para a criação de soluções práticas com foco em sistemas multinúcleo e sua evolução em direção a sistemas de *exascale*. É bem útil para a implementação de serviços de armazenamento de dados com topologias arbitrárias, como as *redes de área de armazenamento*, as quais interligam vários discos conectados em rede que podem ser acessados por quaisquer processos ou aplicações em nuvem que se comunicam através de armazenamento compartilhado. Como um caminho futuro do trabalho, consideramos a possibilidade de construção de algoritmos de consenso genéricos *wait-free* que utilizam outros tipos de registradores e oráculos.

Referências

Aguilera, M. K. (2004). A pleasant stroll through the land of infinitely many creatures. *SIGACT News*, 35(2):36–59.

Aguilera, M. K., Englert, B., and Gafni, E. (2003). On using network attached disks as shared memory. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, PODC '03, pages 315–324, New York, NY, USA. ACM.

Chandra, T. and Toueg, S. (1996). Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267.

Chandra, T. D., Hadzilacos, V., and Toueg, S. (1996). The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722.

Chockler, G. and Malkhi, D. (2002). Active disk paxos with infinitely many processes. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, PODC '02, pages 78–87, New York, NY, USA. ACM.

Delporte-Gallet, C. and Fauconnier, H. (2009). Two consensus algorithms with atomic registers and failure detector omega. In *Proceedings of the 10th International Conference on Distributed Computing and Networking*, ICDCN '09, pages 251–262, Berlin, Heidelberg. Springer-Verlag.

Delporte-Gallet, C., Fauconnier, H., Guerraoui, R., Hadzilacos, V., Kouznetsov, P., and Toueg, S. (2004). The weakest failure detectors to solve certain fundamental problems in distributed computing. In *Proc.s of the XXIII Symp. on Principles of Distributed Computing*, pages 338–346, St. John's, Canada.

Fischer, M. J., Lynch, N. A., and Paterson, M. D. (1985). Impossibility of distributed consensus with one faulty process. *Journal of ACM*, 32(2):374–382.

Gafni, E. and Lamport, L. (2003). Disk paxos. *Distrib. Comput.*, 16(1):1–20.

Goodson, G. R., Wylie, J. J., Ganger, G. R., and Reiter, M. K. (2004). Efficient byzantine-tolerant erasure-coded storage. In *Proc. of the Int. Conf. on Dependable Systems and Networks*, DSN '04, pages 135–144. IEEE Computer Society.

Guerraoui, R. and Raynal, M. (2003). The information structure of indulgent consensus. *IEEE Transactions on Computers*, 53:2004.

Guerraoui, R. and Raynal, M. (2007). The alpha of indulgent consensus. *Comput. J.*, 50(1):53–67.

Herlihy, M. and Luchangco, V. (2008). Distributed computing and the multicore revolution. *SIGACT News*, 39(1):62–72.

Keidar, I. and Rajsbaum, S. (2001). On the cost of fault-tolerant consensus when there are no faults - a tutorial. Technical report.

Khouri, C. and Greve, F. (2013). Algoritmo de consenso genérico em memória compartilhada. XIV WTF-SBRC, Brasília, DF. SBC.

Khouri, C., Greve, F., and Tixeuil, S. (2013). Consensus with unknown participants in shared memory. In *Reliable Distributed Systems (SRDS), 2013 IEEE 32nd International Symposium on*, pages 51–60.

Kubiatowicz, J., Bindel, D., Chen, Y., Czerwinski, S., Eaton, P., Geels, D., Gummadi, R., Rhea, S., Weatherspoon, H., Weimer, W., Wells, C., and Zhao, B. (2000). Oceanstore: an architecture for global-scale persistent storage. *SIGPLAN Not.*, 35(11):190–201.

Lamport, L. (1986). On interprocess communication. *Distributed Computing*, 1(2):77–101.

Lamport, L. (1998). The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169.

Lo, W.-K. and Hadzilacos, V. (1994). Using failure detectors to solve consensus in asynchronous shared-memory systems (extended abstract). In *Proceedings of the 8th International Workshop on Distributed Algorithms*, WDAG '94, pages 280–295, London, UK. Springer-Verlag.

Malkhi, D. and Reiter, M. K. (2000). An architecture for survivable coordination in large distributed systems. *IEEE Trans. on Knowl. and Data Eng.*, 12(2):187–202.

Neiger, G. (1995). Failure detectors and the wait-free hierarchy (extended abstract). In *14th ACM symp. on Principles of distributed computing*, PODC '95, pages 100–109, New York, NY, USA. ACM.

Rodrigues, R. and Liskov, B. (2004). Rosebud: A scalable byzantine-fault-tolerant storage architecture. Technical report, MIT-LCS-TR-932, MIT Laboratory for Computer Science.

TOP500 (2014). Top500 list. *Disponível em http://www.top500.org/lists/2014/11/.*

Zhou, L., Schneider, F. B., and Van Renesse, R. (2002). Coca: A secure distributed online certification authority. *ACM Trans. Comput. Syst.*, 20(4):329–368.