Algoritmo de Consenso Genérico em Memória Compartilhada

Cátia Khouri[1], **Fabíola Greve**[1]

[1]Departamento de Ciências Exatas – Univ. Estadual do Sudoeste da Bahia
Vitória da Conquista, BA – Brasil

[2]Departamento de Ciência da Computação – Univ. Federal da Bahia (UFBA)
Salvador, BA – Brasil

catia091@dcc.ufba.br, fabiola@dcc.ufba.br

Abstract. *Consensus is a fundamental problem for the development of reliable distributed systems. However, in asynchronous environments prone to failures, it is necessary to extend the system with some mechanism that provides the minimum synchrony necessary to circumvent the impossibility of consensus. In this paper, we present a generic consensus algorithm for asynchronous system with shared memory that can be instantiated with a failure detector $\diamondsuit\mathcal{S}$ or Ω. The algorithm is optimal regarding the number of registers it uses and it tolerates $(n-1)$ failures. This solution for shared memory favors the use of consensus in modern applications developed, for example, on multicore architectures and Storage Area Networks (SAN).*

Resumo. *O consenso é um problema fundamental para o desenvolvimento de sistemas distribuídos confiáveis. Porém, em ambientes assíncronos sujeitos a falhas, é preciso estender o sistema com algum mecanismo que forneça o sincronismo mínimo necessário para contornar a impossibilidade do consenso. Neste artigo, apresentamos um algoritmo de consenso genérico para um sistema assíncrono com memória compartilhada que pode ser instanciado com um detector de falhas $\diamondsuit\mathcal{S}$ ou Ω. O algoritmo é ótimo quanto ao número de registradores que utiliza e tolera $(n-1)$ falhas. Essa solução para memória compartilhada favorece o uso do consenso em aplicações modernas desenvolvidas, por exemplo, sobre arquiteturas multicore e Storage Area Networks (SAN).*

1. Introdução

Sistemas distribuídos tolerantes a falhas devem continuar a prover serviços a despeito de falhas em seus nós ou canais de comunicação. É fundamental que mesmo com a falha de alguns participantes, os processos que operam corretamente concordem com relação a determinada informação, para manter a integridade do sistema. É o caso, por exemplo, de sistemas de bancos de dados onde vários processos devem decidir se uma transação deve ser efetivada ou cancelada. Geralmente, quando um processo executa sua computação local com sucesso, posiciona-se favorável à efetivação; caso contrário, manifesta-se pelo cancelamento. Os vários processos então coordenam suas ações para chegar a um *acordo*.

Dentre os problemas de acordo, o *consenso* [Chandra and Toueg 1996, Lo and Hadzilacos 1994] é o mais importante. Ele pode ser visto como uma arcabouço geral de acordo e a maneira mais natural de encapsular esse problema. Informalmente,

o problema do consenso diz respeito a um conjunto de processos que devem concordar sobre um valor (ou conjunto de valores). O problema do consenso está no coração de protocolos como os de sincronização, difusão, reconfiguração, replicação de dados, leitura de sensores e outros. Entretanto, é bem sabido que não existe solução determinística para este problema em sistemas puramente assíncronos sujeitos a falhas, nos modelos de passagem de mensagens e de memória compartilhada [Fischer et al. 1985, Loui and Abu-Amara 1987]. Esse resultado tem motivado o surgimento de abordagens alternativas, enriquecendo-se o sistema com suposições adicionais de sincronia.

Considerando que essa impossibilidade se dá pela dificuldade em se distinguir, num sistema assíncrono, se um processo falhou ou se está muito lento, [Chandra and Toueg 1996] propõem estender o sistema com *detectores de falhas não confiáveis*. Estes se constituem de módulos distribuídos, cada um associado a um processo, que dão dicas sobre processos falhos no sistema, as quais podem ser corretas ou não. Independente disso, algoritmos de consenso corretos são desenhados para sistemas assíncronos estendidos com esses detectores. [Chandra and Toueg 1996] apresentam o detector $\mathcal{W}$ como a classe mais fraca que possibilita resolver o consenso em sistemas assíncronos, nos quais os processos se comunicam através de troca de mensagens, desde que a maioria dos processos seja correta. Através de uma técnica de redução, eles mostram também que as classes de detectores $\diamondsuit\mathcal{S}$ e Ω são equivalentes a $\mathcal{W}$, representando, portanto, requisitos mínimos para resolução do consenso nesse modelo de sistema.

Vários algoritmos têm sido propostos para o modelo assíncrono estendido com detectores de falhas. A maioria deles considera a comunicação baseada em troca de mensagens. Entretanto, dada a importância desse modelo e analogias entre sistemas de passagens de mensagens e de memória compartilhada, alguns estudos foram dedicados a investigar o problema do consenso em sistemas de memória compartilhada ([Lo and Hadzilacos 1994], [Guerraoui and Raynal 2007], [Delporte-Gallet and Fauconnier 2009]). Informalmente, um sistema com memória compartilhada é um conjunto de processos executando que se comunicam através de um conjunto de células de memória compartilhadas sobre as quais há operações que podem ser executadas por um ou mais processos e que representam a única forma de acessá-las. Para cada célula existe um conjunto de valores possíveis de se armazenar.

Neste trabalho apresentamos um algoritmo genérico para o consenso em sistemas assíncronos de memória compartilhada com processos sujeitos a falhas por colapso (*crashing*). O algoritmo é genérico no sentido de que pode ser instanciado com um detector de falhas que pode ser da classe $\diamondsuit\mathcal{S}$ ou Ω, os quais são os detectores mais fracos que permitem realizar o consenso nesse modelo de sistema. Considerando, portanto, os requisitos mínimos de sincronia necessários para resolver o consenso, nosso algoritmo é ótimo. Ele é ótimo também com relação à resiliência, pois tolera qualquer número de falhas. Ao contrário do que ocorre com o modelo de troca de mensagens, em que uma maioria de corretos é exigida, nosso algoritmo permite que um processo correto termine a execução independente do comportamento dos demais, isto é, ele é *wait-free*.

Recentes avanços na tecnologia de armazenamento apontam para sistemas como SAN–*Storage Area Networks* ou *commodity disks* [Aguilera et al. 2003, Guerraoui and Raynal 2007], nos quais os discos, ao invés de serem controlados por um único processo, são ligados diretamente a uma rede de alta velocidade e acessados dire-

tamente pelos clientes. Em alguns sistemas distribuídos, processos se comunicam através desses discos que implementam assim uma memória compartilhada. Como esses discos são mais baratos do que computadores, são uma opção cada vez mais atrativa para se atingir tolerância a falhas e motivam a proposição de algoritmos como o apresentado aqui, apropriado para tal aplicação. O modelo de memória compartilhada também é comum nas máquinas *multicore* atuais, onde processadores compartilham uma única memória física, ou em sistemas distribuídos onde parte da memória de cada processador (e.g., registradores) é compartilhada por vários processos. Assim, o algoritmo proposto se constitui numa ferramenta fundamental para o desenvolvimento de sistemas confiáveis em tais ambientes.

O resto do artigo está estruturado da seguinte maneira: A Seção 2 traz o modelo do sistema. A Seção 3 apresenta o algoritmo genérico de consenso, cujas provas estão na Seção 4. Na Seção 5 é feita uma discussão sobre o algoritmo proposto e trabalhos relacionados. A Seção 6 conclui o artigo.

2. Preliminares

2.1. Modelo do Sistema

Considera-se um sistema distribuído que consiste de um conjunto finito de $n > 1$ processos $\Pi = \{p_1, ..., p_n\}$, dos quais, f podem falhar por colapso (*crashing*), i.e., parando prematura ou deliberadamente. Após parar, um processo não se recupera. Um processo que não falha em uma execução é dito *correto* e um processo que falha é dito *faltoso*. Os processos trocam informações através da leitura e escrita num arranjo $R[n]$ de registradores *regulares* compartilhados do tipo 1-*escritor/n-leitores* (1W/nR), que se comportam corretamente. Um registrador compartilhado modela uma forma de comunicação persistente onde o emissor é o escritor, o receptor é o leitor, e o estado do meio de comunicação é o valor do registrador. Comportamento *correto* de um registrador significa que ele sempre pode executar uma leitura ou escrita e nunca corrompe seu valor.

Um registrador *regular* é mais fraco do que um registrador *atômico* [Lamport 1986]. Num registrador regular, uma leitura não concorrente com uma escrita, retorna o último valor escrito no registrador, e uma leitura que sobrepõe uma ou mais escritas pode obter o último valor escrito antes da leitura iniciar-se ou o valor escrito por qualquer uma das escritas concorrentes. Ou seja, um registrador regular está sujeito a *inversão de valores*. Já um registrador atômico não permite tal inversão. Em um sistema com registradores atômicos, para qualquer execução existe alguma forma de ordenar totalmente leituras e escritas de modo que o valor retornado por uma leitura que sobrepõe uma ou mais escritas seja o mesmo que seria retornado se não houvesse sobreposição.

Não fazemos suposições sobre o tempo que dura cada operação de leitura ou escrita ou mesmo sobre a velocidade dos processos, i.e., o sistema é *assíncrono*. No restante deste artigo, o modelo de sistema definido nesta seção será denominado $\mathcal{AS}[\Pi, f]$.

2.2. Consenso

O problema do consenso é fundamental no projeto de sistemas distribuídos confiáveis. Neste, cada processo p_i propõe um valor v_i e todos os processos têm que decidir por um mesmo valor v. Formalmente, o problema é definido pelas propriedades: (1) Validade – se um processo decide por um valor v, então v é o valor inicial de algum processo;

(2) Acordo uniforme – se um processo decide por um valor v então todos os processos corretos decidem pelo mesmo valor v; (3) Terminação – todos os processos corretos em algum momento acabam por decidir algum valor.

2.3. Detectores de Falhas

Já é bem estabelecido que não existe solução determinística para o consenso em um sistema assíncrono sujeito a falhas [Fischer et al. 1985]. Informalmente, essa impossibilidade é explicada pela dificuldade em distinguir se um processo parou ou está muito lento. Uma solução alternativa é estender o sistema com mecanismos detectores de falhas. Um detector de falhas ($\mathcal{D}$) não confiável é constituído de módulos distribuídos que tem o objetivo de prover o sistema com dicas sobre falhas de processos [Chandra and Toueg 1996]. Cada processo possui um módulo local que funciona como um oráculo fornecendo-lhe, quando requisitado, uma lista de processos considerados suspeitos. Esses módulos podem cometer erros incluindo em suas listas de suspeitos processos corretos ou deixando de incluir processos faltosos. A despeito dessa não-confiabilidade, algoritmos de consenso corretos têm sido propostos para sistemas assíncronos estendidos com esses oráculos [Lo and Hadzilacos 1994, Guerraoui and Raynal 2003].

2.3.1. Detector de Falhas $\diamondsuit\mathcal{S}$

[Chandra and Toueg 1996] classificam os detectores de falhas de acordo com as propriedades completude e acurácia, que os detectores em cada classe exibem. A propriedade completude requer que o $\mathcal{D}$ de fato venha a suspeitar de todo processo faltoso enquanto que a acurácia restringe as suspeições errôneas sobre processos corretos. Os autores então combinam duas propriedades de completude e quatro de acurácia e apresentam oito classes distintas de detectores de falhas. Neste trabalho, o interesse se coloca sobre a classe de detectores de falhas *forte após um tempo* (do inglês, *eventually strong*), ou $\diamondsuit\mathcal{S}$. Um detector $\diamondsuit\mathcal{S}$ satisfaz às seguintes propriedades:

(1) completude forte (*strong completeness*). A partir de algum instante no tempo, todo processo que colapsa será considerado permanentemente suspeito por todo processo correto.
(2) acurácia fraca após um tempo (*eventual weak accuracy*). Existe um instante no tempo a partir do qual algum processo correto jamais será considerado suspeito por qualquer processo correto.

Vale ressaltar que os instantes de estabilidade garantidos pelas propriedades acima não são conhecidos pelos processos. Entretanto, a existência desses instantes permite garantir o término dos algoritmos de consenso baseados em detectores da classe $\diamondsuit\mathcal{S}$.

2.3.2. Detector de Líder Ω

O *detector de líder após um tempo*, conhecido como Ω, também funciona como um oráculo distribuído. O módulo local do detector num processo p_i fornece a identidade de um único processo p_j que p_i considera correto naquele instante. O detector Ω satisfaz à seguinte propriedade:

liderança eventual: existe um instante após o qual o detector fornece a identidade do mesmo processo correto no sistema (i.e., o mesmo líder) para todos os demais processos.

Os detectores $\diamondsuit\mathcal{S}$ e Ω possuem o mesmo poder computacional e são as classes mais fracas de detectores que permitem resolver o consenso em redes assíncronas com Π conhecido, tanto no modelo de passagem de mensagens ([Chandra et al. 1996]), quanto no modelo com memória compartilhada ([Delporte-Gallet et al. 2004]).

3. Algoritmo Genérico para Consenso em Memória Compartilhada

Nesta seção apresentamos um algoritmo genérico para resolver o consenso em um sistema distribuído com memória compartilhada. O algoritmo é genérico porque pode ser instanciado para um sistema estendido com o detector de falhas $\diamondsuit\mathcal{S}$ ou com o detector de líder Ω. Para tanto, o algoritmo define a função $\mathsf{PROPOSITION}_{\mathcal{D}}(e_i, r_i)$ que, a partir de uma consulta ao detector apropriado, tem como objetivo definir a proposta de p_i.

O algoritmo executa em rodadas assíncronas. Cada processo p_i tem acesso às propostas dos demais através da memória compartilhada e tenta tomar uma decisão com base nessas propostas. Entretanto, p_i só decide por um valor v se todas as propostas na memória compartilhada forem iguais a v. Caso contrário, ele parte para uma nova rodada de definição de proposta.

3.1. Memória Compartilhada

A memória compartilhada é constituída de um arranjo $R[n]$ de registradores *regulares* do tipo 1-*escritor-n-leitores* ($1WnR$). O registrador $R[i]$ pertence ao processo p_i, que é o seu único escritor. Entretanto, p_i tem acesso de leitura a qualquer registrador $R[j]$ pertencente a p_j. Cada registrador é composto dos seguintes campos:

(i) $R[i].round$: inteiro que indica a rodada executada por p_i; inicializado com 0.
(ii) $R[i].value$: valor que pode representar uma estimativa, uma proposta ou a decisão do processo; inicializado com $\perp$, denotando um valor padrão que não pode ser proposto por algum processo.
(iii) $R[i].tag$: rótulo que qualifica o valor armazenado em $R[i].value$: est – estimativa; pro – proposta, ou dec – decisão. Inicializado com $\perp$, indica que $R[i].value$ ainda não foi formalizado como estimativa, proposta ou decisão.

As operações que os processos realizam sobre os registradores são as seguintes:

$read(R[i], aux)$: lê o registrador $R[i]$, retornando valor para a variável registro local aux.

$write(R[i], aux)$: escreve o valor na variável registro local aux no registrador $R[i]$.

Embora uma escrita seja executada sobre um registrador (todos os campos), para facilitar a leitura dos algoritmos, algumas vezes expressamos uma escrita de apenas parte dos campos do registrador. Isso é feito sem perda de generalidade, pois como o escritor de cada registrador é único, ele sempre sabe qual foi o último valor escrito. Por exemplo, se a intenção é alterar apenas o campo $value$ de $R[i]$, escrevendo o valor v nele, podemos utilizar a sentença $write(R[i].value, v)$, para denotar a operação:

$$write(R[i].\langle round, value, tag\rangle\ ,\ \langle R[i].round, v, R[i].tag\rangle)$$

3.2. Descrição do Algoritmo de Consenso

O Algoritmo 1 apresenta a função CONSENSUS(v_i) executada por cada processo p_i para decidir por um valor entre os propostos. A entrada v_i é o valor inicial de p_i. O algoritmo funciona em rodadas assíncronas numeradas a partir de 0 (linha 1). O número da rodada é atualizado pela função PROPOSITION$_{\mathcal{D}}(e_i, r_i)$ e depende do oráculo utilizado. Inicialmente, p_i configura sua estimativa para assumir seu valor inicial (linha 1). Então, p_i entra no laço onde permanecerá até que decida um valor (linha 7). Como ao final de alguma iteração do laço a estimativa de p_i pode ser ajustada para $\perp$ (linha 9), ele armazena seu valor corrente de estimativa em v_i para que possa resgatá-lo na próxima iteração(linha 3).

Algorithm 1 CONSENSUS(v_i)

```
(1) r_i := 0;   e_i := v_i;
(2) repeat forever
(3)     if (e_i = ⊥) then e_i := v_i else v_i := e_i;
(4)     PROPOSITION_D(e_i, r_i);
(5)     read (R[1..n] , aux[1..n]);
(6)     proposes := { aux[j].value , ∀ j | aux[j].tag = pro };
(7)     case { (proposes = {v})        then  write (R[i] , ⟨ r_i , v , dec⟩); return (v);
(8)          (proposes = { v , ⊥ })    then  e_i := v; write (R[i] , ⟨ r_i , v , pro⟩);
(9)          (proposes = { ⊥})         then  e_i := ⊥; } endcase
```

A função PROPOSITION$_{\mathcal{D}}(e_i, r_i)$ (linha 4) escreve a proposta de p_i em $R[i]$. p_i então lê o arranjo de registradores (linha 5) e armazena na variável conjunto $proposes$ todas as propostas ali existentes (linha 6). Conforme mostrado adiante, cada $R[i]$ só pode conter, como proposta ($R[i].tag = pro$): um certo valor v, igual para todos os que conseguirem fazer uma proposta válida; ou $\perp$, se o processo não conseguir fazer uma proposta válida. Então, p_i poderá: (i) decidir v, se este for o valor de todas as propostas, e retornar (linha 7); (ii) assumir v como sua estimativa e proposta para a próxima rodada, caso haja propostas iguais a v e a $\perp$, e iniciar a próxima iteração (linha 8); ou (iii) assumir $\perp$ como estimativa para a próxima rodada, caso não haja qualquer proposta válida (linha 9), e assim retomar a estimativa da última rodada (linha 3).

3.3. A Função PROPOSITION$_{\mathcal{D}}(e_i, r_i)$

A função PROPOSITION$_{\mathcal{D}}(e_i, r_i)$ pode ser instanciada com um detector $\diamondsuit\mathcal{S}$ ou Ω. Ela retorna uma proposta v, igual para todos os processos, ou então $\perp$, indicando que p_i não obteve êxito em elaborar uma proposta válida na rodada corrente. Sempre que chamada por CONSENSUS(v_i), PROPOSITION$_{\mathcal{D}}(e_i, r_i)$ executa uma nova rodada cujo número funciona como um relógio lógico, de modo que $r_i > r_j$ indica que um processo p_i que esteja na rodada r_i está mais adiantado, isto é, já executou mais rodadas do que o processo p_j, que está na rodada r_j. Na primeira invocação o valor de e_i é o valor inicial de p_i.

Nas próximas seções são apresentados algoritmos para a função PROPOSITION$_{\mathcal{D}}(e_i, r_i)$ com cada um dos detectores. Além dos registradores compartilhados, os algoritmos usam as seguintes variáveis locais: r_i – número da rodada sendo executada por p_i; e_i – estimativa de p_i; aux – arranjo de registros; a – registro (utilizado como variável auxiliar para a leitura de um registro; possui os campos $\langle r, v, t\rangle$, correspondentes a $\langle round, value, tag\rangle$); c_i – identidade do coordenador da rodada (PROPOSITION$_{\diamondsuit\mathcal{S}}(e_i, r_i)$); l_i – identidade do líder da rodada (PROPOSITION$_{\Omega}(e_i, r_i)$).

3.3.1. Consenso com $\diamondsuit\mathcal{S}$

O Algoritmo 2 (PROPOSITION$_{\diamondsuit\mathcal{S}}(e_i, r_i)$) é uma instância da função de proposição para resolver o consenso com um detector $\diamondsuit\mathcal{S}$. Funciona em rodadas assíncronas sob o paradigma do coordenador rotativo. Cada rodada possui um único coordenador cuja identidade é função do número da rodada que é incrementado de 1 a cada iteração.

Algorithm 2 PROPOSITION$_{\diamondsuit\mathcal{S}}(e_i, r_i)$

```
(1)   r_i := r_i + 1;
(2)   write (R[i], ⟨r_i, e_i, ⊥⟩);
(3)   c_i := (r_i mod n) + 1;        /*calcula a ID do coordenador do round */
(4)   if (c_i = i) then              /*se p_i é coordenador do round */
(5)     read (R[1..n], aux[1..n]); /* verifica estado do registrador
(6)     if (∃ j | aux[j].r > r_i) then   /*se há alguém mais adiantado
(7)       write (R[i], ⟨r_i, ⊥, pro ⟩); /*p_i propõe nada e abandona rodada
(8)     else  if (∃ j | (aux[j].tag = est ∧ aux[j].value ≠ ⊥) then   /*se ∃ estimativa*/
(9)                 e_i := (aux[j].value | ∀j, k: aux[j].tag = aux[k].tag = est,j >= k)
(10)          write (R[i], ⟨r_i, e_i, est⟩);    /* pi divulga sua estimativa (com tag "est") */
(11)          read (R[1..n], aux[1..n]); /* verifica estado do registrador
(12)          if (∃ j | aux[j].r > r_i) then /*se há alguém mais adiantado
(13)            write (R[i], ⟨ r_i, ⊥, pro ⟩); /*p_i propõe nada e abandona rodada
(14)          else  if (∃ j | (aux[j].tag = pro) ∧ (aux[j].value ≠ ⊥)) then  /*se ∃ proposta*/
(15)                    e_i := (aux[j].value | ∀j, k: aux[j].tag = aux[k].tag = pro,j >= k)
(16)                write (R[i] , ⟨ r_i , e_i , pro⟩);    /*p_i divulga sua proposta (tag="pro")*/
(17)  else     /*se pi não é o coordenador da rodada */
(18)    repeat
(19)      read (R[c_i] , a); /*espera proposta do coordenador da rodada ou suspeita */
(20)    until ((a.r > r_i) ∨ (a.t = pro) ∨ (c_i ∈ D_i));
(21)    if ((a.t = pro) ∧ (a.r = r_i)) then /*se proposta válida do coordenador da rodada */
(22)      e_i := a.v; /*adota proposta do coordenador da rodada */
(23)      write (R[i] , ⟨ r_i , e_i , pro ⟩);
(24)  return;
```

Apenas o coordenador propõe um valor na rodada corrente (os demais processos adotam a proposta do coordenador publicando-a em seus registradores). Entretanto, a não confiabilidade inerente aos detectores $\diamondsuit\mathcal{S}$ possibilita a coexistência de coordenadores distintos. Isto porque se qualquer processo p_i, que não é o coordenador corrente, recebe de $\mathcal{D}_i$ a informação de que o coordenador é suspeito (linha 20), embora este permaneça executando (entre as linhas 5 e 16), p_i deixa o *repeat-until* (linhas 18 a 20), retornando para CONSENSUS sem fazer sua proposta. Se nenhuma decisão é tomada, p_i segue para a próxima iteração progredindo para a próxima rodada. Aí, outro coordenador é definido, o qual pode permanecer executando as linhas 5 a 16 junto com ao coordenador anterior.

Dessa forma, se $\mathcal{D}_i$ suspeita erroneamente de coordenadores que não falharam, novos coordenadores podem ser definidos sucessiva e concorrentemente. Nesse sentido, o algoritmo precisa ser indulgente para com o detector, isto é, deve manter a segurança (*safety*) durante períodos de instabilidade e atingir vivacidade quando o sistema se estabiliza [Guerraoui and Raynal 2003].

Descrição do Algoritmo

Parte 1 – Todos os processos

Cada processo p_i começa ajustando o número da rodada (linha 1) e disponibilizando seus valores de rodada e estimativa no registrador compartilhado $R[i]$ (linha 2). Antes de rotular seu registro com "est", esta estimativa de p_i é considerada inválida ($R[i].tag = \perp$). Em seguida, p_i calcula a identidade do coordenador da rodada r_i (linha 3). A partir daí, dois rumos distintos podem ser tomados na Parte 2 (linha 4).

Parte 2a – Processo coordenador da rodada

Se p_i é o coordenador da rodada, vai tentar impor sua proposta. Para isso, p_i lê primeiro os registradores dos demais processos para saber como está o progresso deles (linha 5). Se houver algum processo mais adiantado que ele, p_i desiste de propor um valor e informa o fato escrevendo $\perp$ e "pro", respectivamente, nos campos $value$ e tag de seu registrador (linhas 6-7). Se, no entanto, p_i está na rodada mais adiantada, verifica se já existe alguma estimativa válida ($\neq \perp$) em algum registrador (linha 8). Se houver, p_i assume como sua esta estimativa. Caso contrário, valida sua própria estimativa e a publica (linhas 8-10).

p_i vai então tentar propor sua estimativa (i.e., estabelecê-la como proposta). Para isso ele busca obter as mesmas garantias que obteve para definir sua estimativa: (i) verifica se há algum processo mais adiantado, o que definirá se p_i persistirá no propósito de propor um valor ou não (linhas 11-13); e (ii) verifica se há proposta em algum registrador, o que definirá se sua proposta será igual a alguma pré-existente ou à sua estimativa (linhas 14-15). Se alcançar a linha 16, p_i divulga sua proposta escrevendo em seu registrador um valor diferente de $\perp$ acompanhado do rótulo "pro". Se não tiver alcançado a linha 16, p_i terá escrito o valor $\perp$ acompanhado do rótulo "pro" na linha 7 ou 13, abrindo mão de fazer uma proposta. Em qualquer um dos casos, após escrever seu registrador, p_i retorna para CONSENSUS. A escrita de $\langle -, -, \text{pro} \rangle$ em $R[c_i]$ libera os não-coordenadores da espera nas linhas 18-20, mesmo que o valor escrito em $R[i].value$ seja $\perp$.

Parte 2b – Demais processos: Não coordenadores

Se p_i não é coordenador, tenta obter a proposta do coordenador (linhas 17-20) até que um dos predicados seja verdadeiro: (i) o coordenador publica uma proposta (válida ou não); ou (ii) o coordenador consta da lista de suspeitos de $\mathcal{D}_i$. Se p_i obteve uma proposta do coordenador, p_i assume como sua a proposta do coordenador, divulga-a (linhas 21-23) e retorna para CONSENSUS.

3.3.2. Consenso com Ω

Para esta versão o oráculo utilizado é um detector de falhas da classe Ω. O algoritmo também funciona em rodadas assíncronas, porém aqui apenas o líder progride para uma próxima rodada. Além disso, cada processo, quando líder, executa rodadas com números distintos. Para isso, a primeira rodada executada por um processo p_i tem número igual a i (a identidade de p_i) e as próximas rodadas são calculadas somando-se n ao número da rodada anterior. O resto do algoritmo segue parecido com a versão que usa $\diamondsuit\mathcal{S}$. As diferenças estão, primeiro, no fato de que o líder, diferentemente do coordenador, não é definido em função do número da rodada, e sim da indicação do oráculo Ω (linha 2).

Segundo, o critério de saída da espera nas linhas 18 a 20, é a indicação pelo oráculo Ω de que o líder (corrente até então), deixou de ser líder.

Algorithm 3 PROPOSITION$_\Omega(e_i, r_i)$

```
(1)  if (r_i = 0) then r_i := i;
(2)  l_i := leader();        /*obtém a ID do líder */
(3)  if (l_i = i) then {     /*se p_i é o líder */
(4)     r_i := r_i + n;   write (R[i], ⟨r_i, e_i, ⊥⟩);
(5)     read (R[1..n], aux[1..n]);
(6)     if (∃ j | aux[j].r > r_i) then  /*se há alguém mais adiantado, p_i propõe nada /abandona
(7)        write (R[i], ⟨r_i, ⊥, pro ⟩);
(8)     else  if (∃ j | (aux[j].tag = est ∧ aux[j].value ≠ ⊥) then  /*se ∃ estimativa*/
(9)              e_i := (aux[j].value | ∀j, k: aux[j].tag = aux[k].tag = est, j >= k)
(10)          write (R[i], ⟨r_i, e_i, est⟩);   /* pi divulga sua estimativa (com tag "est") */
(11)          read (R[1..n], aux[1..n]);
(12)          if (∃ j | aux[j].r > r_i) then
(13)             write (R[i], ⟨ r_i, ⊥, pro ⟩);
(14)          else  if (∃ j | (aux[j].tag = pro) ∧ (aux[j].value ≠ ⊥)) then  /*se ∃ proposta*/
(15)                   e_i := (aux[j].value | ∀j, k: aux[j].tag = aux[k].tag = pro, j >= k)
(16)                write (R[i] , ⟨ r_i , e_i , pro⟩);   /*p_i divulga sua proposta (tag="pro")*/
(17) else     /*se pi não é o líder */
(18)    repeat
(19)       read (R[l_i] , l);
(20)    until ((a.r > r_i) ∨ (a.t = pro) ∨ (l_i ≠ leader()));
(21)    if (a.t = pro) then
(22)       e_i := a.v;
(23)       write (R[i] , ⟨ r_i , e_i , pro ⟩);
(24) return;
```

Como no caso dos não coordenadores, os processos não líderes tentam obter a proposta do líder (linhas 17-20), mas aqui a repetição é abandonada se Ω informa uma identidade de processo diferente da informada no início da função (linha 2).

Note que a exemplo do algoritmo anterior, aqui é possível a coexistência de diversos líderes, uma vez que o detector Ω pode retornar identidades distintas para invocações distintas. Da mesma forma, o algoritmo precisa ser indulgente para com o detector, garantindo a segurança (*safety*) durante períodos de instabilidade e atingir vivacidade quando o sistema se estabiliza.

4. Prova de Corretude do Algoritmo de Consenso Genérico

Notação. Considere a seguinte notação para as provas do Consenso: Um processo "p_i propõe um valor v" quando p_i escreve o valor v, bem como o tag = *pro* no seu registrador $R[i]$, ou seja, quando o comando **write** $(R[i], \langle -, v, \text{pro} \rangle)$ é executado por p_i.

Lema 1 *Em um sistema $\mathcal{AS}[\Pi, f]$ estendido com $\mathcal{D}_{\diamond\mathcal{S}}$ ou $\mathcal{D}_\Omega$, se algum processo p_i que invoca PROPOSITION$_\mathcal{D}(-, r_i)$ propõe $v \neq \perp$, então algum processo p_j invocou PROPOSITION$_\mathcal{D}(v, r_j)$, $r_j \leq r_i$.*

Prova. A partir de uma inspeção nos Algoritmos 2 e 3, vemos que um valor $v \neq \perp$ só pode ser proposto por um processo p_i: (A) pelo processo coordenador (linha 16); ou (B)

por um processo não coordenador (linha 23). Neste último caso, o processo não coordenador simplesmente assume a proposta imposta pelo coordenador. Então, é suficiente provar o caso (A). A proposta escrita no registrador (linha 16) é o valor armazenado em e_i. Inicialmente, esse valor é passado como argumento na invocação da função e pode permanecer inalterado até a escrita da proposta. Entretanto, e_i pode ser atualizado antes:

(i) se algum processo p_j tiver escrito uma estimativa em uma rodada anterior $r_j < r_i$ (linhas 8-9). Nesse caso, e_i recebe este valor (ou alguma das estimativas, se houver mais de uma). Note que a primeira estimativa escrita no registrador é, forçosamente, um valor passado como argumento na invocação da função, já que nessa situação, o processo avalia o predicado da linha 8 como $falso$;

(ii) se algum processo p_j tiver escrito uma proposta numa rodada anterior $r_j < r_i$ (linhas 14-15). Nesse caso, e_i recebe este valor (ou alguma das propostas, se houver mais de uma). Note que a primeira proposta escrita no registrador por um processo p_j é, forçosamente, um valor passado como argumento na chamada da função (seja por p_j ou por outro processo), pois se o predicado da linha 14 é avaliado $falso$, a proposta será o próprio e_j recebido na invocação ou uma estimativa obtida conforme (i).

De (A) e (B), o lema segue. □

Lema 2 *Em um sistema $\mathcal{AS}[\Pi, f]$ estendido com $\mathcal{D}_{\Diamond\mathcal{S}}$ ou $\mathcal{D}_{\Omega}$, se algum processo p_i que invoca PROPOSITION$_{\mathcal{D}}(-,-)$ propõe $v \neq \perp$, então $\forall p_j$ que invoca PROPOSITION$_{\mathcal{D}}(-,-)$ e propõe algum valor, propõe $w = v \vee w = \perp$.*

Prova. Vamos considerar especificamente o caso da função PROPOSITION$_{\Diamond\mathcal{S}}$. A prova para o caso da função PROPOSITION$_{\Omega}$ é análoga a esta, bastando fazer a transposição coordenador/líder e das respectivas linhas citadas do algoritmo. Considere dois casos em que um processo p_i faz uma proposta (escreve no registrador $\langle r_i, v, \text{pro}\rangle$):

CASO 1. p_i não é coordenador da rodada. Nesse caso p_i apenas assume a proposta do coordenador (linhas 18 a 23) e portanto o lema se confirma.

CASO 2. p_i é coordenador da rodada. Podemos desmembrar esse caso em dois:

CASO 2A. p_i é o único coordenador da rodada e executa sozinho as linhas 4-16. Trivialmente, se houver proposta válida em algum registrador, é assumida por p_i (linhas 14-15).

CASO 2B. p_i é coordenador da rodada, mas há outros coordenadores: p_i executa o trecho das linhas 4-16 concorrentemente com os demais coordenadores. Isso pode acontecer devido à não confiabilidade de $\mathcal{D}_{\Diamond\mathcal{S}}$ (veja seção 3.3.1). Sem perda de generalidade, suponha que um coordenador p_j executa concorrentemente com p_i e que p_i é o primeiro a propor $v \neq \perp$. Suponha, por contradição, que p_j propõe $w \neq v$. Se ambos propõem valores, ambos executam a linha 16 do algoritmo. Considere os seguintes instantes de tempo:

t_0: instante em que p_i inicia a leitura dos registradores da linha 5;
t_1: instante em que p_j inicia a escrita de $R[i]$ da linha 2;
t_2: instante em que p_j inicia a leitura dos registradores da linha 5; logo $t_1 < t_2$ (A)
t_3: instante em que p_i inicia a escrita de $R[i]$ da linha 10;
t_4: instante em que p_i inicia a leitura dos registradores da linha 11; logo $t_3 < t_4$ (B)

(1) Para p_i propor o valor v na linha 16, pode-se concluir que o predicado da linha 6 foi avaliado *falso*, logo, quando p_i executou a leitura de R[i] da linha 5, p_j ainda não

havia executado a escrita da linha 2 ou, dada a regularidade dos registradores, ambos executaram a leitura e escrita concorrentemente; portanto, temos que $t_0 \leq t_1$.

(2) p_i divulgou sua estimativa executando a escrita de R[i] na linha 10 com $e_i = v$.

(3) Se p_j, pela nossa hipótese, propõe $w \neq v$, na linha 16, então p_j não pode ter atribuído v a e_j na linha 9, isto é, p_j avaliou o predicado da linha 8 como *falso*, donde se conclui que p_j realizou a leitura da linha 5 antes ou concorrentemente à escrita de p_i da linha 10; logo, $t_2 \leq t_3$.

(4) Para que p_i não executasse a escrita da linha 13 (abandonando a chance de fazer sua proposta), deve ter avaliado o predicado da linha 12 *falso*. Logo, nesse instante, quando p_i fez a leitura da linha 11, não havia (ele não pode ter achado) qualquer processo numa rodada mais adiantada que a dele (linha 11). Se esse fosse o caso, a leitura de R_i por p_i teria que ter iniciado antes da escrita de p_j da linha 2; portanto, nesse caso, $t_4 \leq t_1$

De (A), (3), (B) e (4), temos $t_1 < t_2$; $t_2 \leq t_3$; $t_3 < t_4$; $t_4 \leq t_1$ – uma contradição. □

Lema 3 *Em um sistema $\mathcal{AS}[\Pi, f]$ estendido com $\mathcal{D}_{\diamondsuit\mathcal{S}}$ ou $\mathcal{D}_\Omega$, se todo processo p_i invoca PROPOSITION$_\mathcal{D}(v)$, então todo p_i só pode propor v ou $\perp$.*

Prova. A prova é direta de PROPOSITION$_\mathcal{D}$ e do Lema 1. Propostas só podem ser feitas nas linhas 7, 13 ou 16. Nas duas primeiras, o valor proposto é sempre $\perp$. Na última, o valor proposto é $\neq \perp$. Pelo Lema 1, um valor proposto v é tal que algum processo invocou PROPOSITION$_\mathcal{D}(v, -)$. Logo, se todo processo p invoca PROPOSITION$_\mathcal{D}(v, -)$, v é o único valor $\neq \perp$ que pode ser proposto. □

Lema 4 *Em um sistema $\mathcal{AS}[\Pi, f]$ estendido com $\mathcal{D}_{\diamondsuit\mathcal{S}}$ ou $\mathcal{D}_\Omega$, todo processo correto que invoca PROPOSITION$_\mathcal{D}(-,-)$ continuamente, em algum instante propõe um valor $v \neq \perp$.*

Prova. Para este lema, vamos considerar o caso da função PROPOSITION$_{\diamondsuit\mathcal{S}}(e_i, r_i)$. A prova para o caso da função PROPOSITION$_\Omega(e_i, r_i)$ é análoga a esta, bastando fazer a transposição coordenador/líder, considerando as respectivas linhas do algoritmo. Além disso, deve-se considerar a propriedade de liderança eventual de Ω.

A cada rodada é definido um coordenador em função do número da rodada (linha 3). Enquanto o coordenador tenta impor uma proposta (linhas 5 a 16), os não coordenadores aguardam para obter a proposta do coordenador (linhas 18 a 20). Uma vez que o coordenador escreva sua proposta no registrador (linha 16), os demais processos obtém e assumem essa proposta (linhas 19 a 23). Entretanto, se algum processo suspeita que o coordenador falhou (linha 20), retorna da função sem fazer uma proposta (linhas 21-24).

Da propriedade acurácia fraca após um tempo do detector de falhas $\diamondsuit\mathcal{S}$, existe um instante a partir do qual algum processo correto jamais será considerado suspeito por qualquer processo correto. Então, existe um instante a partir do qual um processo correto que é coordenador fará uma proposta v (linha 16) e os demais processos corretos obterão e assumirão essa proposta (linhas 19 a 23). □

Lema 5 *Em um sistema $\mathcal{AS}[\Pi, f]$ estendido com $\mathcal{D}_{\diamondsuit\mathcal{S}}$ ou $\mathcal{D}_\Omega$, ao invocar CONSENSO$(-)$, se processo p_i decide por um valor v, então qualquer processo p_j que decide, decide v.*

Prova Acordo uniforme: Para que um processo p_i decida na linha 7, ele tem que ter encontrado todas as propostas no arranjo de registradores iguais a v (linhas 5-7). Pelo Lema 2, se algum processo que invoca PROPOSITION$_{\mathcal{D}}(-,-)$ propõe um valor $v \neq \perp$, então todos os demais processos que invocam PROPOSITION$_{\mathcal{D}}(-,-)$ e propõem um valor, propõem v ou $\perp$. Logo, nenhum processo p_j irá propor e escrever no seu registrador um valor w, $v \neq w \neq \perp$. Portanto, se p_j decide na linha 7, ele decide v. □

Lema 6 *Em um sistema $\mathcal{AS}[\Pi, f]$ estendido com $\mathcal{D}_{\diamondsuit\mathcal{S}}$ ou $\mathcal{D}_{\Omega}$, ao invocar CONSENSO$(-)$, um processo correto p_i termina por decidir um valor v.*

Prova Terminação: O processo p_i decide ao executar a linha 7. Pelo Lema 2, se um processo propõe v, todas as propostas diferentes de $\perp$ são iguais a v. Pelo Lema 4, todo processo correto que invoca PROPOSITION$_{\mathcal{D}}(-,-)$ continuamente, em algum instante propõe um valor $v \neq \perp$. Então, todo processo correto p_i acaba por satisfazer o predicado da linha 7 e decide v. □

Lema 7 *Em um sistema $\mathcal{AS}[\Pi, f]$ estendido com $\mathcal{D}_{\diamondsuit\mathcal{S}}$ ou $\mathcal{D}_{\Omega}$, ao invocar CONSENSO$(-)$, se um processo p_i decide v, então v é o valor inicial de algum processo.*

Prova Validade: Os únicos valores de entrada no algoritmo são os valores iniciais dos processos passados na invocação de CONSENSUS(v_i) por p_i. Esses mesmos valores são passados para PROPOSITION$_{\mathcal{D}}(e_i, r_i)$ em sua primeira invocação (linha 4). Então, qualquer estimativa e proposta escrita nos registradores na chamda de PROPOSITION$_{\mathcal{D}}(e_i, r_i)$ é uma cópia de um desses valores iniciais (linhas 8-10, 14-16). Assim, um valor v decidido é um dos valores propostos inicialmente por algum processo (linhas 5-7).

Teorema 1 *Em um sistema $\mathcal{AS}[\Pi, f]$ estendido com $\mathcal{D}_{\diamondsuit\mathcal{S}}$ ou $\mathcal{D}_{\Omega}$, o Algoritmo 1 satisfaz às propriedades do Consenso (definidas em 2.2).*

Prova. Segue diretamente dos lemas 5, 6 e 7. □

Teorema 2 *O Algoritmo 1 é wait-free.*

Prova. Mostramos que o algoritmo é correto a despeito de $(n-1)$ falhas. Suponha um instante t em uma execução em que $(n-1)$ processos falham. Seja p_i o único processo que não falha nesta execução. Ao executar PROPOSITION$_{\mathcal{D}}(-,-)$, em algum instante p_i virá a ser coordenador (ou líder) e não abandona precipitadamente sua execução sem fazer proposta, já que não haverá qualquer processo em uma rodada maior (linhas 6-7 e 12-13). Pelo contrário, virá a propor um valor v e ao retornar, decidirá por este valor. □

5. Discussão e Trabalhos Relacionados

Considerando que os detectores $\diamondsuit\mathcal{S}$ ou Ω são os mais fracos que permitem o consenso, o algoritmo proposto é ótimo com relação aos requisitos de sincronia. Ele também é ótimo com relação à resiliência, uma vez que ele é *wait-free* – admite (n-1) faltas, qualquer processo termina em um número limitado de passos, independente do comportamento dos demais. Quanto ao custo, o algoritmo também é ótimo [Lo and Hadzilacos 1994]. São necessários apenas n registradores regulares, do tipo 1-*escritor-n-leitores*. Esses registradores são mais fracos, por um lado, do que registradores atômicos, e por outro, do que registradores *n-escritores-n-leitores*. Além disso, se o detector ($\diamondsuit\mathcal{S}$ ou Ω) se comporta perfeitamente, na primeira rodada o algoritmo converge.

O problema do consenso é amplamente estudado e vários algoritmos têm sido propostos, a maioria deles para sistemas em que os processos se comunicam através de passagem de mensagens [Chandra and Toueg 1996, Guerraoui and Raynal 2003]. Para o modelo de memória compartilhada, poucos são os trabalhos identificados, além do nosso [Lo and Hadzilacos 1994, Delporte-Gallet et al. 2004, Guerraoui and Raynal 2007, Khouri et al. 2012].

[Lo and Hadzilacos 1994] propõem um algoritmo de consenso para um sistema assíncrono. Entretanto, eles utilizam registradores *atômicos*, enquanto que nós utilizamos registradores regulares, que são mais fracos. Eles provam que são necessários pelo menos n registradores (atômicos) 1-*escritor/n-leitores* ($1WnR$) para construir algoritmos de consenso *wait-free*, usando um detector de falha da classe *Strong*. O nosso algoritmo funciona corretamente com n registradores $1WnR$ mais fracos – regulares, usando um detector mais fraco ($\diamondsuit\mathcal{S}$ ou Ω).

[Guerraoui and Raynal 2003] identificam a estrutura da informação dos algoritmos de consenso indulgentes para com seus oráculos e estudam a complexidade inerente a estes. Eles apresentam um algoritmo de consenso genérico que pode ser instanciado com um oráculo específico e mantém a mesma complexidade segundo alguns critérios. Eles consideram um modelo assíncrono sujeito a falhas, onde os processos se comunicam através de passagem de mensagens e o número máximo de falhas deve ser $f < n/2$.

[Guerraoui and Raynal 2007] apresentam um arcabouço que unifica uma família de algoritmos de consenso baseado no detector de falhas Ω. O algoritmo pode ser configurado para modelos de comunicação diferentes: memória compartilhada, rede de área compartilhada, passagem de mensagens e sistemas de discos ativos, mas em qualquer desses modelos, o oráculo considerado é o Ω.

Trilhando um caminho semelhante ao de [Guerraoui and Raynal 2003, Guerraoui and Raynal 2007], nós propomos um algoritmo genérico que permite configurar o detector de falhas a ser utilizado–$\diamondsuit\mathcal{S}$ ou Ω, em um modelo de memória compartilhada. Enquanto o algoritmo proposto em [Guerraoui and Raynal 2003] é restrito a sistemas com $f < n/2$ (número de processos faltosos inferior à metade do número de participantes), nós mostramos que no modelo de memória compartilhada, é possível obter algoritmos indulgentes com uma resiliência $n - 1$.

[Khouri et al. 2012] propõem um protocolo para consenso num sistema dinâmico (o conjunto de processos é desconhecido). Sua abordagem consiste em usar a abstração *detector de participantes* para construir o *membership* do sistema e, conforme o grafo da conectividade do conhecimento, aplicar um algoritmo clássico para a realização do consenso. Com uma pequena adaptação para contar o número de processos participantes durante o conhecimento do sistema, o algoritmo aqui apresentado pode ser utilizado neste protocolo e resolver o consenso num sistema dinâmico de memória compartilhada.

6. Conclusão

Neste artigo apresentamos um algoritmo genérico para a resolução do consenso num sistema assíncrono sujeito a falhas que pode ser instanciado com um detector $\diamondsuit\mathcal{S}$ ou Ω. O algoritmo proposto pode ser aplicado a sistemas distribuídos em que processadores compartilham parte de sua memória entre vários processos, máquinas *multicore* atuais, ou

sistemas como *Storage Area Networks*, utilizados para o compartilhamento de armazenamento e que implementam uma memória compartilhada.

Mostramos que é possível construir tal algoritmo, para o modelo proposto, usando n registradores regulares. Este é o número mínimo de registradores atômicos necessários definido na literatura [Lo and Hadzilacos 1994] para qualquer algoritmo de consenso *wait-free* em um modelo estendido com um detector mais forte que o adotado por nós. Consideramos para um próximo trabalho investigar a possibilidade de construir algoritmos de consenso *wait-free* utilizando apenas registradores *safe*, os quais retornam o valor lido, se a leitura não sobrepõe alguma escrita; porém podem retornar qualquer valor do domínio se a leitura for concorrente a uma ou mais escritas [Lamport 1986].

References

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.

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 *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, PODC '04, pages 338–346, New York, NY, USA. ACM.

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.

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.

Khouri, C., Greve, F., and Tixeuil, S. (2012). Consenso com participantes desconhecidos em memoria compartilhada. 30o SBRC, Porto Alegre, RS, Brasil. SBC.

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

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.

Loui, M. C. and Abu-Amara, H. H. (1987). Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, 4:163–183.