UNIVERSIDADE FEDERAL DA BAHIA

TESE DE DOUTORADO

CONSENSO EM MEMÓRIA COMPARTILHADA DINÂMICA

Cátia Mesquita Brasil Khouri

Programa de Pós-Graduação em Ciência da Computação

Salvador
15 de maio de 2015

PGCOMP-DSc-2015

CÁTIA MESQUITA BRASIL KHOURI

CONSENSO EM MEMÓRIA COMPARTILHADA DINÂMICA

Esta Tese de Doutorado foi apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal da Bahia, como requisito parcial para obtenção do grau de Doutor em Ciência da Computação.

Orientadora: Profa. Dra. Fabíola Gonçalves Pereira Greve

Salvador
15 de maio de 2015

Sistema de Bibliotecas - UFBA

Khouri, Cátia Mesquita Brasil
Consenso em memória compartilhada dinâmica / Cátia Mesquita Brasil Khouri. - 2015.
68f.:il.

Orientadora: Profª. Drª. Fabíola Gonçalves Pereira Greve
Tese (doutorado) - Universidade Federal da Bahia, Instituto de Matemática, Salvador, 2015.

1. Sistemas operacionais distribuídos (Computadores). 2. Consenso (Computadores). 3. Tolerância a Falhas (Computadores). I. Greve, Fabíola Gonçalves Pereira. II. Universidade Federal da Bahia. Instituto de Matemática. III Título.

CDD - 005.1
CDU - 004

TERMO DE APROVAÇÃO

CÁTIA MESQUITA BRASIL KHOURI

CONSENSO EM MEMÓRIA COMPARTILHADA DINÂMICA

Esta Tese de Doutorado foi julgada adequada à obtenção do título de Doutor em Ciência da Computação e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação da Universidade Federal da Bahia.

Salvador, 15 de maio de 2015

Profa. Dra. Fabíola Gonçalves Pereira Greve
Universidade Federal da Bahia

Profa. Dra. Luciana Bezerra Arantes
University of Pierre et Marie Currie

Prof. Dr. Eduardo Adilio Pelinson Alchieri
Universidade de Brasília

Prof. Dr. George Marconi de Araújo Lima
Universidade Federal da Bahia

Prof. Dr. Lásaro Jonas Camargos
Universidade Federal de Uberlândia

DEDICATÓRIA

Não dá pra dizer que construir uma tese é uma tarefa fácil. Na verdade, fazer um curso de doutorado é uma empreitada um tanto custosa... ainda mais quando já não se tem 20 ou 30 anos... Talvez seja possível para alguém, mas, concluído o trabalho, a maior certeza que eu tenho é a de que não consegui sozinha. Muita gente esteve envolvida na obra, principalmente aqueles a quem a dedico − minha família!

Quando digo família, quero dizer F A M Í L I A !! A começar pelo Criador, pai amoroso, o maior e mais justo compartilhador de recursos que já conheci. Cada vez mais me asseguro de que ele me ama e cuida de mim.

Meus pais, João e Iracema. Sua calorosa acolhida em Salvador me fizeram lembrar, por muitas vezes, os velhos tempos de criança. Que pena, pai, que você não viu o desfecho da história! Logo você, quem mais acreditou que ela chegaria a um final feliz! Mãe, você nem imagina como tem sido importante o simples fato de você existir. Amo vocês!

Walter Júnior, meu marido, companheiro, que com seu carinho e cuidado (às vezes, excessivo!) me ajudou a caminhar até o fim. Seu amor, inteligência, sabedoria e poesia me inspiram!

Davi e Daniel, filhos queridos, par perfeito. Vocês são o melhor presente que ganhei do Pai. Junto com seu pai, meus maiores professores. Não posso negar que a confiança que vocês têm em mim, às vezes me assombra, mas por outro lado me incentiva e faz sentir capaz. Poly, querida, nossa casa ficou mais alegre com a sua chegada!

Walter Khouri e Zaia, importantes referências de amor, carinho e família. Cuidaram de meus filhos nas tantas vezes que estive fora (e continuam cuidando até hoje!).

Cássia, Cristiane, Gal, Nadia, Luisinho, Deda, Trigo, Mauro, Mara, Jairão, Ivan, Lindsen, Isaque, André, João, Pedro, Raquel, Samuel, Noelle, tios e primos. Vocês me ajudam a encontrar meu lugar no grande sistema distribuído que é esse mundo de meu Deus.

AGRADECIMENTOS

Agradeço a Deus por ter me dado mais este presente dentre tantos no curso da vida.

À Profa. Dra. Fabíola Greve, por sua orientação cuidadosa, disponibilidade, comentários, sugestões, incentivo, humanidade. Aprendi muito em cada uma de nossas reuniões. Obrigada por compartilhar seu conhecimento comigo!

Aos colegas da Área de Computação do DCET/UESB, por apoiar meu afastamento e por "segurar as pontas" enquanto estive fora.

Aos colegas do DMCC, em especial, Marcão, pelas discussões e companheirismo.

À Coordenação do DMCC, professores e funcionários, por sua dedicação e pronto atendimento em todas as vezes que recorri a eles.

À UESB, por todo apoio, inclusive financeiro, que viabilizou minha trajetória até aqui.

Tudo que você tem não é seu
Tudo que você guardar
Não lhe pertence nem nunca lhe pertencerá
...
Só é seu aquilo que você dá
...
E o beijo que você deu é seu, é seu, é seu o beijo...
— LAMPIRÔNICOS (Pop Zen)

Assim, não importa o que eu diga, no que eu creia ou o que eu faça: sem amor, estou falido.

— A MENSAGEM - BÍBLIA EM LINGUAGEM CONTEMPORÂNEA
I Carta aos Coríntios 13:3

RESUMO

Sistemas distribuídos modernos, construídos sobre redes móveis ad-hoc, entre pares (P2P), grades oportunistas ou nuvens, permitem que seus participantes acessem serviços e informações independente de sua localização ou topologia da rede. Garantir tais serviços exige um projeto de sistema confiável completamente distribuído e que possa lidar com o dinamismo, falhas e falta de conhecimento global.

Esta tese dedica-se ao estudo de dois problemas fortemente relacionados e que são fundamentais no desenvolvimento de sistemas distribuídos confiáveis: o problema do consenso e o problema da eleição de líder após um tempo, considerando um ambiente assíncrono e dinâmico, em que os processos entram e saem do sistema aleatoriamente, se comunicam através de uma memória compartilhada e podem falhar por parada.

A maioria das propostas para esses problemas foca em sistemas estáticos onde o conjunto de participantes é conhecido e fixo, e a comunicação se dá por passagem de mensagens. Entretanto, classes importantes de sistemas como os serviços centrados em dados tolerantes a falhas e altamente disponíveis, além das máquinas com arquitetura multinúcleo, onde os processos compartilham uma única memória física, não se adequam a esse modelo. No primeiro caso, algumas aplicações relevantes são as redes de área de armazenamento (SANs); sistemas de armazenamento P2P; e sistemas de passagem de mensagem nos quais servidores são modelados como componentes de armazenamento.

Nesta tese, é proposto um conjunto de algoritmos para o problema do consenso tolerante a falhas em sistemas assíncronos com memória compartilhada, onde o conjunto de participantes é desconhecido. Duas abordagens são exploradas. A primeira considera a abstração detector de participantes que auxilia na construção do conhecimento do sistema. A solução conta com um algoritmo de consenso genérico baseado em oráculo que, ao que se sabe, é o primeiro algoritmo de consenso para memória compartilhada que pode ser instanciado com um detector de falhas da classe $\diamondsuit\mathcal{S}$, ou um detector de líder da classe Ω. A segunda apresenta um consenso genérico com outra característica inovadora: não depende do conhecimento da cardinalidade do conjunto de participantes, de modo que suporta o dinamismo do sistema.

Ainda na direção de prover serviços fundamentais para ambientes dinâmicos de memória compartilhada, propõe-se um protocolo de detecção de líder após um tempo. O algoritmo é livre de tempo, no sentido de que não usa temporizadores para garantir a convergência. Ao invés disso, baseia-se num padrão de acesso à memória compartilhada. Ao que se sabe, é o primeiro algoritmo livre de tempo a implementar um serviço de líder Ω para memória compartilhada.

Na prática, um consenso genérico e modular é um arcabouço que permite construir sistemas dinâmicos de camadas superiores independentes do detector de falhas que está disponível. Nesse caso, a implementação do consenso pode ser melhor adaptada às car-

acterísticas particulares de cada ambiente; principalmente quando a implementação do detector serve a muitas aplicações. Desse modo, as aplicações existentes que já estão executando sobre os detectores $\diamondsuit\mathcal{S}$ podem ser portadas mais facilmente.

Palavras-chave: Sistemas distribuídos. Sistemas distribuídos dinâmicos. Memória compartilhada. Consenso. Tolerância a falhas. Detector de líder.

ABSTRACT

Modern distributed systems, built on ad-hoc mobile networks, peer to peer (P2P), opportunistic grids or clouds, allow participants to access services and information regardless of their location or network topology. Ensuring such services requires a fully distributed reliable system design and that can handle the dynamism, failures and lack of global knowledge.

This thesis is dedicated to the study of two closely related problems, both of which are fundamental for the development of reliable distributed systems: the problem of consensus and the problem of eventual leader election considering an asynchronous and dynamic environment in which the processes join and leave the system at will, communicate through a shared memory and may fail to stop.

Most proposals for said problems are focused on static systems where the number of participants is known and fixed, and communication happens by message passing. However, important classes of systems as fault-tolerant and highly available data centers, in addition to machines with multi-core architecture, in which processes share a single physical memory, do not fit into that model. In the first case, some relevant applications are storage area networks (SANs); P2P storage systems; and message passing systems in which servers are modeled as storage components.

In this thesis, we propose a set of algorithms to the fault tolerant consensus problem in asynchronous systems with shared memory, where the number of participants is unknown. Two approaches are explored. The first considers the abstraction detector participants that aids the system knowledge building. The solution has a generic consensus algorithm based on oracle, far as is known, it is the first consensus algorithm for shared memory that can be instantiated with a failure detector of the $\diamondsuit\mathcal{S}$class, or a leader detector of the Ω class. The second shows a generic agreement with another innovative feature: it does not depend on knowledge of the cardinality of the set of participants so that the system supports dynamic.

Still in the direction of providing basic services to dynamic environments of shared memory, it is proposed a eventual leader detection protocol. The algorithm is free-time, in the sense that it does not uses timers to guarantee convergence. Instead, it is based on a shared memory access pattern. From what is known, it is the first time-free algorithm to implement a Ω leader service to shared memory.

In practice, a generic and modular consensus is a framework that allows you to build upper layers dynamic systems independent of the failure detector that is available. In this case, the implementation of the consensus may be better adapted to the particular characteristics of each environment; especially when the implementation of the detector serves many applications. Thus, existing applications that are already running on the $\diamondsuit\mathcal{S}$ detectors can be more easily ported.

Keywords: Distributed systems. Dynamic distributed systems. Shared memory. Consensus. Fault tolerance. Leader detector.

SUMÁRIO

Capítulo 1

Este capítulo apresenta o objeto de estudo da tese, sua motivação e justificativa, os objetivos perseguidos, a metodologia utilizada, os resultados parciais alcançados e a estrutura do documento.

INTRODUÇÃO

Sistemas distribuídos tolerantes a falhas devem continuar a prover serviços de acordo com a especificação, 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 possam colaborar de alguma maneira. Em muitas situações, os processos livres de falhas precisam concordar com relação a uma determinada informação para manter a integridade do sistema. Uma importante abstração em sistemas distribuídos tolerantes a falhas é o *acordo*.

Dentre os problemas de acordo, o *consenso* (FISCHER; LYNCH; PATERSON, 1985) é o mais importante. Ele pode ser visto como um arcabouço geral de acordo e a maneira mais natural de encapsular esse problema. Dado um conjunto de processos, o consenso é um problema em que cada processo propõe um valor e todos têm que decidir por um mesmo valor entre os propostos.

O problema do consenso está no coração de protocolos como os de sincronização, gestão da filiação (*group membership*), confirmação atômica (*atomic commitment*), difusão, replicação em máquinas de estados e muitos outros (PAUL et al., 1985; GUERRAOUI; SCHIPER, 2001). A complexidade envolvida nestas situações normalmente pode ser abstraída através de uma primitiva *consenso*. A importância do problema do consenso, portanto, é devida à sua onipresença e por isso mesmo este problema tem sido alvo de inúmeros estudos teóricos e práticos nas últimas décadas.

Apesar da simplicidade de seu enunciado, o consenso está longe de ser um problema de fácil resolução. É bem sabido que não existe solução determinística para o problema do consenso em sistemas puramente assíncronos em que pelo menos um processo possa falhar por parada (*crashing*) (FISCHER; LYNCH; PATERSON, 1985; LOUI; ABU-AMARA, 1987). Intuitivamente, esta impossibilidade se dá porque, como não há limites predefinidos para o tempo em que um processo dá um passo ou para o tempo que leva entre a emissão e a recepção de uma mensagem, é difícil distinguir se um processo falhou ou se apenas está muito lento.

A assincronia em um sistema distribuído dá origem a um fator de incerteza que dificulta o projeto de algoritmos determinísticos (quando não impossibilita, como é o caso do consenso). Uma alternativa para contornar esta dificuldade é estender o sistema com um detector de falhas não confiável (CHANDRA; TOUEG, 1996) ou um detector de líder (CHANDRA; HADZILACOS; TOUEG, 1996), os quais funcionam como oráculos distribuídos. Um detector de falhas, quando requisitado por um processo, fornece uma lista de processos considerados falhos. Já um detector de líder, retorna ao processo que o invoca, a identidade de um processo considerado correto. Após um período de tempo finito, a identidade do mesmo processo correto é retornada para cada processo correto. As informações fornecidas por esses oráculos podem estar equivocadas em certos momentos, mas mesmo assim é possível escrever algoritmos corretos contando com essas informações (GUERRAOUI, 2000). As classes de detectores de falhas, $\diamondsuit\mathcal{S}$, e de líder, Ω, quando o conjunto de participantes é conhecido, são equivalentes e fornecem as condições minimais, em termos de sincronismo, que permitem resolver o consenso.

De fato, várias abordagens têm sido propostas na literatura para o problema do consenso, porém a maioria delas considera um modelo de comunicação interprocessos baseado em passagem de mensagens e em sistemas em que o conjunto de processos participantes é conhecido *a priori*. Contudo, dada a importância do modelo de memória compartilhada, alguns estudos têm se voltado para o consenso nesse modelo de comunicação (DELPORTE-GALLET; FAUCONNIER, 2009; ASPNES, 2012; RAYNAL; STAINER, 2012; HERLIHY; RAJSBAUM; RAYNAL, 2013). Um sistema de memória compartilhada compreende um conjunto de processos que se comunicam através de uma memória comum, constituída de *objetos compartilhados*. Um *objeto compartilhado* é definido por um tipo e um conjunto de operações primitivas que são a única forma de acessá-lo. Além da memória compartilhada, os processos também podem ter uma memória privada local.

Ocorre que a maioria dos estudos encontrados na literatura para o modelo de memória compartilhada considera que o conjunto de participantes é conhecido e fixo. Entretanto, uma infinidade de sistemas distribuídos modernos tais como aqueles desenvolvidos para redes de sensores, dispositivos móveis *ad hoc*, grades oportunistas, redes entre pares (P2P) ou computação em nuvem, permite que processos entrem e saiam arbitrariamente do sistema. Nesses ambientes, chamados *dinâmicos*, o conjunto de participantes não é conhecido *a priori* e é constantemente alterado durante a execução de uma computação distribuída. Esse dinamismo introduz uma incerteza adicional nos sistemas assíncronos que agrava a dificuldade no projeto de algoritmos.

Estes sistemas apresentam um desenho que dispensa a existência de uma autoridade administrativa centralizada ou uma infraestrutura estática. Garantir tal serviço requer um projeto de sistema completo que possa lidar adequadamente com dinamismo, falhas e assincronismo. O problema do consenso, portanto, é fundamental, também nesse contexto, uma vez que os nós precisam coordenar suas ações para garantir consistência.

O modelo assíncrono de memória compartilhada condiz com as modernas arquiteturas multinúcleo, onde vários núcleos acessam uma mesma memória física. A chegada e a hegemonia dessas máquinas é irreversível. O estado da arte atual aponta para um cenário em que desde supercomputadores até simples sensores serão multiprocessadores com memória compartilhada (HERLIHY; 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 (o computador que ocupa a primeira posição na última lista Top500 conta com $3,12 \times 10^3$ núcleos atingindo $33,86$ petaflops (TOP500, 2014)). O gerenciamento de recursos nesses ambientes exige a disponibilidade de serviços de coordenação e acordo das mais variadas naturezas. Confiabilidade e tolerância a falhas figuram como importantes desafios nessa área.

Esses sistemas representam um vasto campo de aplicação para protocolos de consenso e eleição de líder como os apresentados nesta tese. Em particular, Ω tem sido aplicado como gerenciador de contenção (AGUILERA; ENGLERT; GAFNI, 2003; FERNÁNDEZ et al., 2010) e foi mostrado por (GUERRAOUI; KAPALKA; KOUZNETSOV, 2006) que Ω é o gerenciador de contenção mais fraco que permite transformar memória transacional em software livre de obstrução (*obstruction-free*) em uma memória transacional não bloqueante (*nonblocking*) (HERLIHY; LUCHANGCO; MOIR, 2003; HERLIHY et al., 2003).

Outra classe importante de sistemas é a de serviços de armazenamento de dados de alta disponibilidade e tolerantes a falhas, tais como as redes de área de armazenamento (ou SANs – *Storage Area Networks*) (AGUILERA; ENGLERT; GAFNI, 2003). Em uma SAN, um conjunto de discos conectados a uma rede (ou NAD – *Network Attached Disks*) pode ser acessado diretamente por qualquer processo no sistema através de requisições de escrita e leitura. Desse modo, um NAD pode ser visto como uma memória compartilhada onde os blocos de discos são 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 esse motivo, protocolos para a implementação de serviços como consenso e detecção de líder, desenvolvidos para sistemas dinâmicos de memória compartilhada, podem ser mapeados para tais sistemas de armazenamento. As vantagens desta estrutura têm motivado o projeto de algoritmos de consenso baseados em disco (AGUILERA; ENGLERT; GAFNI, 2003; CHOCKLER; MALKHI, 2002; GAFNI; LAMPORT, 2003; GUERRAOUI; RAYNAL, 2007), que são capazes de fornecer serviços de armazenamento distribuídos confiáveis.

O problema do consenso em sistemas de memória compartilhada sujeitos a falhas por parada de processos foi anteriormente investigado em (LO; HADZILACOS, 1994; GUERRAOUI; RAYNAL, 2007; DELPORTE-GALLET; FAUCONNIER, 2009; HERLIHY; RAJSBAUM; RAYNAL, 2013) e descobriu-se que as condições necessárias e suficientes nem sempre coincidem com as de sistemas de troca de mensagens. Em particular, enquanto neste último é necessário que a maioria dos processos seja correta para resolver o consenso, as soluções no modelo de memória compartilhada geralmente podem ser livres de espera (HERLIHY, 1991), ou seja, o protocolo permanece correto, mesmo que todos os processos, exceto um, falhem. Mas até onde se sabe, as soluções existentes em memória compartilhada levam em conta um modelo clássico onde se tem conhecimento completo sobre a composição do sistema e a sua cardinalidade.

Uma abordagem comum para a solução do problema do consenso em sistemas onde o conjunto de participantes é desconhecido consiste em tratá-lo em duas fases distintas – a

primeira refere-se à determinação do conjunto de participantes; a segunda diz respeito à resolução do consenso propriamente dito. Para cumprir a primeira etapa, uma alternativa é o uso da abstração *detector de participantes* (CAVIN; SASSON; SCHIPER, 2004; GREVE; TIXEUIL, 2007), que fornece dicas a respeito dos nós que participam correntemente no sistema. A partir dessas dicas é possível construir o *grafo da conectividade do conhecimento* cujas características permitem definir condições de conectividade suficientes e necessárias para a realização do consenso em ambientes assíncronos estendidos com algum mecanismo de detecção de falhas. No Capítulo 3 considera-se esta abordagem.

Outra abordagem consiste em delegar ao sistema de memória compartilhada subjacente o gerenciamento da conectividade do sistema em face às entradas e saídas de participantes. Nos capítulos 4 e 5 esta alternativa é adotada.

Considerando o destaque que atualmente se dá aos sistemas dinâmicos de memória compartilhada e a importância de serviços como consenso e eleição de líder para a manutenção da consistência em tais ambientes é que decidiu-se investigar soluções para estes problemas. Questões como as seguintes nortearam as investigações:

1. É possível escrever um algoritmo genérico para o consenso em um sistema de memória compartilhada estendido com um detector de falhas sem que se tenha que definir *a priori* este detector?

2. É possível escrever um protocolo para o consenso em um sistema dinâmico de memória compartilhada estendido com um oráculo detector de falhas ou de líder e com a ajuda de um detector de participantes?

3. Em tal protocolo, é necessário conhecer o número de participantes *a priori*?

4. Qual o oráculo adequado nesta situação?

5. Existe um algoritmo para implementar tal oráculo?

Dado que na literatura não foram encontradas respostas a estas questões, as investigações conduzidas resultaram nesta tese que foi assim construída de maneira incremental, perseguindo os objetivos descritos a seguir.

1.1 OBJETIVOS

Os objetivos estabelecidos para este trabalho são direcionados a dois problemas fortemente relacionados, considerando um ambiente assíncrono dinâmico em que os processos se comunicam através de uma memória compartilhada e podem falhar por parada: estudar o problema do consenso tolerante a falhas e o problema de implementar um detector de líder em tal ambiente. Mais especificamente, busca-se:

1.1.1 Definir um modelo adequado para abordar o problema do consenso em ambientes dinâmicos

Para se propor alguma solução em um sistema distribuído, é necessário caracterizar devidamente o modelo de sistema em questão. Uma vez que não há solução determinística

para o consenso em sistemas assíncronos sujeitos a falhas, suposições adicionais precisam ser feitas com relação às condições de sincronismo do sistema de modo a possibilitar uma solução. Da mesma forma, é preciso considerar condições de resiliência bem como o modelo de memória compartilhada no que diz respeito aos tipos de objetos disponíveis, quais processos têm acesso aos objetos e através de quais operações.

O sincronismo adicional necessário ao sistema assíncrono para contornar a impossibilidade do consenso pode ser encapsulado em abstrações como detectores de falhas não confiáveis e detectores de líder (CHANDRA; TOUEG, 1996; CHANDRA; HADZILACOS; TOUEG, 1996). Esses mecanismos funcionam como oráculos distribuídos que fornecem dicas sobre falhas de processos.

Neste trabalho são exploradas as classes de detectores de falhas *forte após um tempo* (*eventually strong* ou $\diamondsuit\mathcal{S}$) e de *líder após um tempo* (*eventual leader* ou Ω). Deve-se ressaltar que o uso desses detectores em um ambiente dinâmico com memória compartilhada precisa ser melhor avaliado e ajustes precisam ser feitos, seja na especificação das classes, seja nas condições em que os detectores poderão ser implementados no sistema.

1.1.2 Desenvolver um conjunto de algoritmos para resolução do consenso em um ambiente dinâmico com memória compartilhada

Importantes classes de sistemas como serviços de centros de dados altamente disponíveis e tolerantes a falhas funcionam sobre redes dinâmicas onde a comunicação se dá através de uma memória compartilhada. Diferentemente dos sistemas clássicos, onde o conjunto de participantes e suas identidades são conhecidos, em ambientes dinâmicos não se sabe *a priori* quais são os pares com quem se pode colaborar, nem quantos deles estão disponíveis. Recentemente, condições necessárias e suficientes foram identificadas para a resolução do consenso nesse contexto, mas tomando-se por base o modelo de comunicação por passagem de mensagens.

Desta maneira, torna-se necessário desenvolver um conjunto de protocolos que com a ajuda da abstração detector de participantes e, atendidas as condições de conectividade mínimas necessárias, habilite os processos a atingir o consenso a partir da execução de um algoritmo apropriado. A ideia é que esse algoritmo seja genérico com relação ao oráculo de modo que possa ser executado independente de qual oráculo esteja disponível na camada subjacente. Uma vez que não foi encontrado na literatura um algoritmo de consenso genérico para memória compartilhada, seu desenvolvimento inclui-se no conjunto de algoritmos propostos na tese.

1.1.3 Desenvolver um algoritmo para implementação de Ω

O detector Ω figura como uma abstração fundamental em aplicações tolerantes a falhas. Ele representa uma forma elegante de encapsular o sincronismo necessário à resolução de problemas cruciais como consenso, efetivação atômica (atomic commitment), implementação de registrador atômico, etc. (DELPORTE-GALLET et al., 2004). Dada a sua importância, um objetivo da tese consiste em desenvolver um protocolo para implementação de Ω no ambiente dinâmico considerado. Tal algoritmo é necessário também para completar a solução para o consenso.

Todas as implementações de Ω para ambientes assíncronos de memória compartilhada encontradas na literatura baseiam-se em temporizadores para garantir terminação, isto é, consideram que certas suposições temporais sobre os sistemas são satisfeitas ao cabo de um tempo. Assim, para o algoritmo aqui concebido, são levantadas suposições outras sobre o modelo do sistema que não implicam no uso de temporizadores.

1.2 METODOLOGIA E ESTRUTURA DA TESE

O desenvolvimento das atividades passou por várias etapas. A revisão da literatura foi um primeiro passo que se estendeu durante todo o trabalho abrangendo temas como sistemas distribuídos, consenso, sistemas dinâmicos, modelo de comunicação baseado em memória compartilhada, modelo de falhas por parada, detectores de participantes, detectores de falhas e detectores de líder. Um resumo do que foi estudado e os principais conceitos e definições utilizados na tese podem ser encontrados no *Capítulo 2*.

Em seguida, foi estabelecido o modelo de sistema a considerar para construir a solução para o consenso tolerante a falhas em um sistema assíncrono com participantes desconhecidos em que a comunicação se dá através de memória compartilhada – SM FT-CUP. O modelo definido foi o de *chegada infinita com concorrência limitada*, segundo o qual o número de processos participantes no sistema é infinito, mas assume-se que em cada execução o número de participantes, n, é finito, embora desconhecido dos processos. Foi então escolhida a classe de detectores de participantes k-OSR e desenvolvido o conjunto de algoritmos apresentado no *Capítulo 3*. A partir daí, sentiu-se a necessidade de desenvolver um algoritmo específico para o consenso que pudesse ser agregado ao conjunto de protocolos.

Para que o algoritmo pudesse ser utilizado com menos dependência do serviço de detecção de falhas oferecido pelo sistema subjacente, decidiu-se que o algoritmo deveria ser genérico, no sentido de poder ser instanciado com um detector qualquer de uma das classes que supre o sistema com as condições minimais para se atingir o consenso, a saber, $\diamondsuit\mathcal{S}$ e Ω. O primeiro algoritmo obtido (duas versões, na realidade), atende à demanda do protocolo SM FT-CUP, mas requer que os participantes conheçam n. Uma nova versão então foi obtida a partir da anterior, em que os processos não precisam do conhecimento prévio de n. Nesse caso, novas suposições foram adicionadas ao modelo para dar conta do dinamismo. A versão final do algoritmo é apresentada no *Capítulo 4*.

Em uma última etapa, a solução foi ampliada com o desenvolvimento de um algoritmo para implementar um detector da classe Ω. Para contornar a impossibilidade devida à assincronia do sistema, buscou-se uma abordagem menos comum do que aquelas baseadas em suposições temporais explícitas. Foi então proposta uma propriedade *time-free* sobre um padrão de acesso à memória compartilhada que junto com a definição de um limite inferior sobre o número de processos estáveis no sistema possibilitaram a implementação do algoritmo. Nesse caso também, uma primeira versão foi desenvolvida e em seguida otimizada com o intuito de reduzir as operações de escrita na memória compartilhada. A versão final com as provas de corretude encontra-se no *Capítulo 5*.

Algumas considerações e conclusões sobre o estudo realizado e os resultados alcançados, bem como sugestões de futuros trabalhos são apresentados no *Capítulo 6*.

1.3 PUBLICAÇÕES

O trabalho desenvolvido resultou, até então, nas seguintes publicações.

1. Consenso com Participantes Desconhecidos em Memória Compartilhada
XXX Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC 2012)
Autores: Cátia Khouri, Fabíola Greve, Sébastien Tixeuil.
Resumo: O consenso é um problema fundamental para o desenvolvimento de soluções confiáveis em sistemas distribuídos dinâmicos, tais como sistemas P2P, grades oportunistas, redes móveis ad-hoc, etc. Diferentemente dos sistemas clássicos, onde o conjunto de participantes e suas identidades são conhecidos, nos sistemas dinâmicos não se sabe a priori quais são os pares com quem se pode colaborar, nem quantos deles estão disponíveis. Recentemente, condições necessárias e suficientes foram identificadas para a resolução do consenso nesse contexto, mas tomando-se por base o modelo de comunicação por passagem de mensagens. Nesse artigo, estendemos tais resultados e apresentamos protocolos para a resolução do consenso com participantes desconhecidos num modelo de memória compartilhada.

2. Algoritmo de Consenso Genérico em Memória Compartilhada
XIV Workshop de Testes e Tolerância a Falhas (WTF/SBRC 2013)
Artigo premiado com o 1^o lugar no WTF 2013.
Autores: Cátia Khouri, Fabíola Greve.
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 network*.

3. Consensus with Unknown Participants in Shared Memory
32nd International Symposium on Reliable Distributed Systems (SRDS 2013)
Autores: Cátia Khouri, Fabíola Greve.
Resumo: The shared memory model matches important classes of systems deployed over dynamic networks, as for example, fault-tolerant and high available data centric services. Consensus is a fundamental building block able to realize such reliable distributed systems. Unlike the classical setting where the full set of participants and their identities are known to every process, dynamic networks preclude such global knowledge to be available. In this paper, we investigate and present protocols to solve fault-tolerant consensus in an environment with unknown participants that communicate *via* shared memory.

4. A Generic Consensus Algorithm for Shared Memory

The 19th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC 2013)

Autores: Cátia Khouri, Fabíola Greve.

Resumo: The shared memory model matches important classes of modern applications, such as fault-tolerant and highly available data centric services. Consensus is an important building block able to realize such reliable distributed systems. However, there exists no deterministic solution to consensus in asynchronous systems prone to failures. Failure and leader detectors are elegant abstractions which encapsulate the extra synchrony necessary to circumvent this impossibility. In this paper, we present a generic consensus algorithm for asynchronous shared memory systems able to be instantiated with two fundamental detectors, namely $\diamondsuit\mathcal{S}$ and Ω. The algorithm is wait-free, tolerating up to $n-1$ failures, and optimal, regarding the synchrony required and the number of registers it uses.

5. Serviço de Líder em Sistemas Dinâmicos com Memória Compartilhada

XIV Workshop de Testes e Tolerância a Falhas (WTF/SBRC 2014)

Artigo premiado com o 1^o lugar no WTF 2014.

Autores: Cátia Khouri, Fabíola Greve.

Resumo: Apesar da importância que ambientes dinâmicos como as SANs (redes de área de armazenamento) e as arquiteturas multinúcleo ocupam no cenário atual dos sistemas distribuídos, poucas são as propostas de modelos e protocolos para a implementação de eleição de líder após um tempo nesses contextos. Essa abstração é fundamental para a implementação dos requisitos de consistência e tolerância a falhas desses sistemas. Entretanto, a maioria das abordagens de eleição de líder considera sistemas estáticos, onde os processos se comunicam por troca de mensagens, satisfazendo requisitos temporais. Este artigo apresenta um serviço de eleição de líder, da classe Ω, para um *sistema dinâmico* assíncrono, sujeito a falhas por parada, em que os processos se comunicam através de *registradores atômicos compartilhados* e segundo um *padrão de acesso à memória*, livre de requisitos temporais.

6. Eleição assíncrona de líder em memória compartilhada dinâmica

XXXIII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC 2015)

Autores: Cátia Khouri, Fabíola Greve.

Resumo: Sistemas distribuídos em nuvens, redes de área de armazenamento (SAN's), etc. são inerentemente dinâmicos. O detector de líder é uma abstração poderosa para garantir a consistência e tolerância a falhas em face de tal dinamismo. Este artigo apresenta um serviço de líder, de classe Ω, para o modelo de memória dinâmica compartilhada, sujeito a falhas de processos. Diferentemente das soluções conhecidas, ele não requer o conhecimento da cardinalidade do conjunto de participantes do sistema, de tal forma que nós pode sair e entrar no sistema livremente.

7. Consenso Genérico em Sistemas Dinâmicos com Memória Compartilhada
XV Workshop de Testes e Tolerância a Falhas (WTF/SBRC 2015)
Autores: Cátia Khouri, Fabíola Greve.
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.

Capítulo

2

Este capítulo traz alguns conceitos e definições fundamentais em sistemas distribuídos, essenciais para a compreensão da tese como um todo. Ao apresentar os conceitos, são citadas referências importantes da literatura. Um estado da arte mais completo a respeito de temas específicos relacionados aos algoritmos produzidos será apresentado nos próximos capítulos, juntamente com os respectivos algoritmos.

CONCEITOS FUNDAMENTAIS

2.1 INTRODUÇÃO

Um sistema distribuído consiste de uma coleção de componentes autônomos, conectados através de uma rede, que se comunicam, coordenam suas ações e compartilham recursos (TANENBAUM; STEEN, 2006; COULOURIS et al., 2011). De modo geral, na literatura referente a algoritmos distribuídos, assim como neste trabalho, esses componentes são abstraídos como *processos* ou *nós*. Um algoritmo distribuído é uma coleção de autômatos idênticos, um por processo. O autômato em um processo define seu comportamento, isto é, os passos de computação executados pelo processo. Uma *execução* de um algoritmo distribuído é representada por uma sequência de passos executados pelos processos envolvidos no algoritmo (CACHIN; GUERRAOUI; RODRIGUES, 2011).

Uma abordagem amplamente utilizada para estudar e caracterizar sistemas distribuídos é considerar os vários tipos de modelos que podem ser derivados a partir de distintas suposições concernentes a propriedades estruturais e comportamentais do sistema. Por exemplo, com relação ao conjunto de processos participantes, pode-se considerar que este conjunto é conhecido ou desconhecido e sua cardinalidade pode ser conhecida ou não, ou mesmo infinita, isto é, o sistema pode crescer com o passar do tempo de maneira ilimitada (AGUILERA, 2004). Várias possibilidades também há quanto à velocidade relativa dos processos e dos eventos referentes à comunicação, quanto à forma como ocorre a comunicação interprocessos; e quanto aos tipos de falhas a que processos e meios de comunicação estão sujeitos.

Assim é que ao se propor soluções para sistemas distribuídos é preciso definir o *modelo* para o qual a solução se aplica. Na realidade, algoritmos distribuídos em geral são desenhados tendo-se em mente determinado modelo de sistema, e a menor alteração em qualquer uma das suposições adotadas pode resultar em sensíveis modificações no algoritmo. Da mesma forma, as condições necessárias e suficientes para se resolver certo

problema podem mudar drasticamente sob as mínimas alterações no modelo do sistema. É o que se verifica no caso do problema do consenso, cuja resiliência varia de acordo com as suposições de sincronismo e de falhas, e até mesmo apresenta solução em determinados modelos enquanto que em outros, não (DOLEV; DWORK; STOCKMEYER, 1987; DWORK; LYNCH; STOCKMEYER, 1988).

Nesta tese, considera-se um modelo de computação distribuída assíncrono, não confiável, em que os processos se comunicam através de uma memória compartilhada. Em tal modelo, a velocidade dos processos é totalmente arbitrária, isto é, cada processo progride em uma velocidade que é variável e independente da velocidade dos demais. Além disso, os processos podem falhar, parando prematuramente. Essas propriedades introduzem um fator de incerteza que se traduz no aumento da dificuldade em lidar com tais sistemas. Essa incerteza é exacerbada quando o conjunto de participantes do sistema não é fixo, como ocorre com os sistemas chamados *dinâmicos*.

Dada a importância dessas dimensões do modelo do sistema – comunicação interprocessos, sincronismo, falhas e configuração do conjunto de processos participantes – apresenta-se a seguir uma breve discussão a seu respeito com o fim de auxiliar no entendimento do restante do texto. Também são apresentados aqui alguns conceitos fundamentais cujos significados são relevantes para a tese como um todo, a exemplo do problema do consenso e medidas de complexidade de algoritmos para sistemas de memória compartilhada. Conceitos mais específicos, concernentes aos protocolos produzidos, bem como o estado da arte, serão vistos juntamente com os algoritmos nos respectivos capítulos.

Para facilitar a apresentação, é assumida a existência de um relógio global, não acessível pelos processos, que pode ser utilizado por um observador externo ao sistema para medir o tempo. Os passos dos processos são executados de acordo com esse relógio: cada passo é executado com um tique do relógio.

2.2 MODELOS DE COMUNICAÇÃO INTERPROCESSOS

Os dois principais modelos de comunicação entre processos em um sistema distribuído são o *modelo de passagem de mensagens* e o *modelo de memória compartilhada*. No primeiro, os processos interagem enviando e recebendo mensagens através dos canais de comunicação. No segundo, a comunicação se dá através de uma memória constituída de objetos que são compartilhados entre os processos (*objetos compartilhados*). Este último é o modelo adotado nesta tese e suas características serão vistas com mais detalhes.

2.2.1 Memória Compartilhada

Um sistema de memória compartilhada consiste de um conjunto de processos que compartilham uma memória comum. No contexto da tese, a memória compartilhada é uma abstração distribuída construída no topo de um sistema de passagem de mensagens que permite aos processos comunicarem-se invocando operações sobre *objetos compartilhados* (HERLIHY; WING, 1990; CACHIN; GUERRAOUI; RODRIGUES, 2011).

Um *objeto compartilhado*, por sua vez, é uma abstração de uma estrutura de dados acessada por vários processos, o qual possui um *tipo*, que define o *domínio de valores*, $\mathbb{D}$,

que seu estado pode assumir, isto é, valores que podem ser armazenados no objeto; e um conjunto de *operações primitivas* que se constituem na única forma de acessar o objeto.

Em muitos aspectos, tais objetos se assemelham àqueles providos por máquinas multiprocessadoras (ou multinúcleo), em nível de hardware ou software, onde vários processadores (ou núcleos) compartilham um único espaço de endereçamento, isto é, uma única memória física. Essas abstrações também apresentam similaridades com um dispositivo de disco acessado através de uma rede de área de armazenamento (ou SAN – *Storage Area Network*) ou um arquivo em um sistema de arquivos distribuído.

No modelo de sistema considerado, objetos compartilhados são implementados por processos que se comunicam trocando mensagens através de uma rede e não compartilham qualquer dispositivo de memória física. Entretanto, dadas as suas semelhanças, algoritmos projetados para sistemas distribuídos de memória compartilhada como os que são apresentados aqui, podem ser aplicados em tais ambientes que compartilham dispositivos de armazenamento. A Figura 2.1 representa um sistema distribuído de memória compartilhada implementado sobre um sistema de passagem de mensagens. Os processos na camada inferior trocam mensagens para implementar os objetos providos à camada superior, na qual os processos se comunicam acessando os objetos compartilhados.

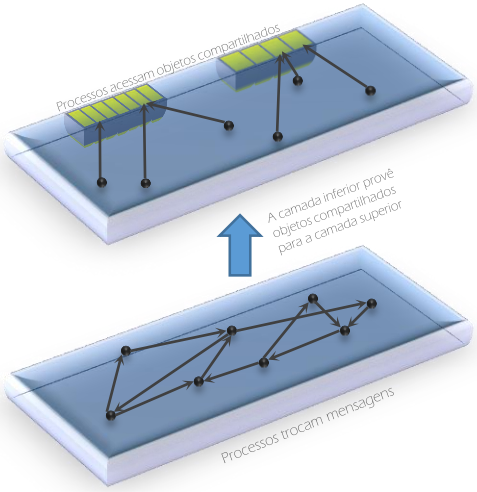

Figura 2.1 Implementação de uma memória compartilhada distribuída.

Em um sistema de memória compartilhada, o algoritmo definido pelo autômato de cada processo consiste de uma sequência de operações locais e operações de acesso aos objetos compartilhados. Normalmente, pelo menos uma das operações primitivas de acesso ao objeto envolve a gravação/escrita de um valor e uma outra envolve a obtenção/leitura do valor correntemente armazenado. Assim, um objeto 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 objeto. No decorrer do texto, um *objeto compartilhado* pode ser referido desta maneira ou simplesmente *objeto*.

Os acessos de um processo p sobre um objeto compartilhado O são sequenciais. Isso significa que se p invoca a operação op^1 sobre O, só virá a invocar outra operação op^2 sobre

o mesmo objeto após ter completado op^1. Uma operação sobre um objeto compartilhado envolve um evento de *invocação*, que ocorre em um instante t_i, e um evento de *resposta*, que ocorre em um instante t_r e delimita o término da operação. Portanto, uma operação sobre um objeto não é instantânea, ela dura um intervalo de tempo desde sua invocação até a resposta, isto é, $[t_i, t_r]$. Com base nesses eventos, é possível definir se duas operações invocadas por processos distintos sobre um mesmo objeto são concorrentes com relação ao tempo real.

Definição 2.1 (Precedência.)**.** *Considere as operações op^1 e op^2, invocadas, respectivamente, nos instantes ${t_i}^1$ e ${t_i}^2$; e que retornam nos instantes ${t_r}^1$ e ${t_r}^2$, respectivamente. Diz-se que op^1 precede op^2 ($op^1 \prec op^2$) se e somente se ${t_r}^1 < {t_i}^2$.*

Definição 2.2 (Concorrência.)**.** *Duas operações op^1 e op^2 são concorrentes ($op^1 \parallel op^2$) se op^1 não precede op^2 e op^2 não precede op^1.*

Uma execução de um sistema de memória compartilhada distribuída é modelada por uma sequência de eventos de invocação e de reposta (HERLIHY; WING, 1990). Uma invocação é dita *pendente* num prefixo de execução se ele não contém a resposta correspondente. Uma execução é *sequencial* se toda invocação, exceto possivelmente a última, é seguida imediatamente pela resposta correspondente.

Uma execução que não é sequencial, é *concorrente*. Uma execução concorrente corresponde a alguma execução sequencial que é definida de acordo com a *propriedade de consistência* assumida. Por exemplo, uma propriedade de consistência mais relaxada pode considerar que duas operações não concorrentes apareçam na execução sequencial correspondente na mesma ordem do tempo real, enquanto que duas operações concorrentes possam aparecer em qualquer ordem.

Esta propriedade de consistência descreve o comportamento de um objeto compartilhado sob acessos concorrentes e, em última análise, a visão que os processos têm deste objeto. Na definição dos objetos específicos mais adiante, as propriedades de consistência são apresentadas seguindo a metodologia utilizada por (BALDONI; BONOMI; RAYNAL, 2009). Para facilitar a apresentação, considera-se, sem perda de generalidade, que cada valor só pode ser escrito uma vez em um objeto. Quando necessário, uma operação de leitura sobre um objeto O e o valor correspondente retornado por ela, v, será denotada $read(O)[v]$.

O objeto compartilhado mais simples que existe é o *registrador* (LAMPORT, 1986), o qual suporta apenas uma operação de escrita e uma de leitura. Alguns objetos mais complexos também são encontrados na literatura, implementados em software ou em hardware, como *set*, *test-and-set*, *semaphores*, *compare-and-swap*, *fetch-and-insert*, etc. (HERLIHY, 1988; AFEK et al., 1995; BALDONI; BONOMI; RAYNAL, 2009). Nesta tese, os algoritmos apresentados utilizam dois tipos de objeto compartilhado: *registrador* e *conjunto* (*set*).

2.2.1.1 Registrador. Um registrador, denotado R, é um tipo de objeto compartilhado que armazena um valor e provê duas operações primitivas (LAMPORT, 1986):

(i) $read(R)$, a qual retorna o estado do registrador R [1]; e

(ii) $write(R, v)$, que altera o estado do registrador R para v e retorna OK.

Pode-se definir alguns tipos de registradores de acordo com o domínio de valores que podem ser armazenados, a forma de acesso (quantos processos podem ter acesso de escrita e quantos de leitura) e ainda conforme o comportamento numa execução concorrente.

Domínio de estados. Um registrador é *binário* se pode assumir um de dois valores: 0 ou 1. Um registrador *multivalorado* pode assumir qualquer valor inteiro.

Direitos de acesso. Quanto ao acesso de processos, são definidas as seguintes classes de registradores:

- 1-*escritor*-1-*leitor* ($1W1R$) – o registrador pode ser escrito por um único escritor e lido por um único leitor;
- 1-*escritor-múltiplos-leitores* ($1WMR$) – o registrador pode ser escrito por um único escritor e lido por múltiplos leitores;
- *múltiplos-escritores-múltiplos-leitores* ($MWMR$) – o registrador pode ser escrito por múltiplos escritores e lido por múltiplos leitores.

Nesta tese só são utilizados registradores conforme definidos por (LAMPORT, 1986), isto é, $1WMR$. Além disso, a menos que se diga o contrário, é assumido que os registradores são inicializados com um valor padrão $\perp$. Isso quer dizer que uma operação de escrita que toma o valor $\perp$ como parâmetro é realizada sobre cada registrador antes de qualquer outra computação.

Consistência. De acordo com o comportamento em execuções concorrentes, um registrador $1WMR$ pode ser *seguro* (*safe*), *regular* ou *atômico* (LAMPORT, 1986). Considerando que cada registrador só pode ser escrito por um processo (o escritor), as operações de escrita sobre um mesmo registrador são sempre sequenciais. Assim, a semântica de um registrador R é definida pelo valor retornado por uma operação $read(R)$. Nas definições a seguir, quando necessário, uma operação de leitura sobre um registrador R e o respectivo valor de retorno, x, serão denotados $read(R)[x]$.

Registrador safe. Um registrador *safe*, R_s, é o que apresenta a semântica mais fraca. Uma leitura não concorrente a uma escrita num registrador *safe* retorna sempre o valor correto, isto é, o último valor escrito. Porém, quando uma leitura é concorrente a uma ou mais operações de escrita, só se pode garantir que o valor retornado é um valor do domínio do registrador, $\mathbb{D}(R_s)$. A leitura pode retornar até mesmo um valor que nunca foi escrito no registrador.

A Figura 2.2 ilustra o comportamento de um registrador *safe* cujo domínio de estados $\mathbb{D} = \{x, y, z\}$. Cada linha horizontal representa o progresso de um processo com a

[1] Nos próximos capítulos utiliza-se a notação $read(R, v)$ para a operação de leitura sobre o registrador R cujo valor obtido é armazenado na variável local v.

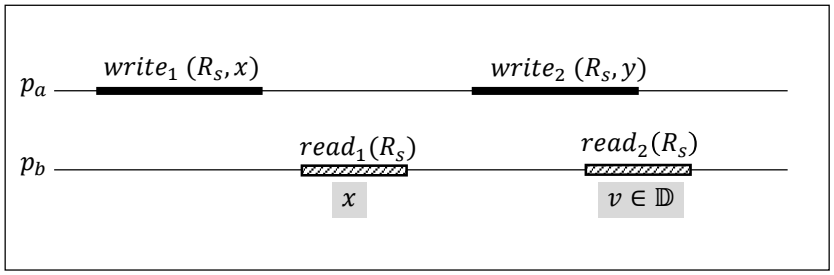

Figura 2.2 Comportamento de um registrador safe R_s. Supondo que $\mathbb{D}(R_s) = \{x, y, z\}$, $read_1(R_s)$ retorna x. $read_2(R_s)$ pode retornar x, y ou z.

evolução do tempo indo da esquerda para a direita. As operações *write* são representadas por barras cheias e as operações *read* por barras hachuradas. Os possíveis valores de retorno em cada operação *read* encontram-se entre colchetes abaixo da barra correspondente. A operação $read_1(R_s)$ retorna o último valor escrito no registrador (pela operação $write_1(R_s, x)$). A operação $read_2(R_s)$ é concorrente à operação de escrita $write_2(R_s, y)$ e portanto pode retornar qualquer valor do domínio de R_s.

Os valores que podem ser retornados por uma operação de leitura num registrador *safe* são definidos a seguir.

Definição 2.3 (Valor admissível para um registrador $safe$: $read(R_s)$)**.** *Dado um registrador $safe$, R_s, uma execução r e uma operação $read(R_s) \in r$, um valor $v \in \mathbb{D}(R_s)$ é admissível para $read(R_s)$ se:*

(i) $\exists\ write(R_s, v) \in r : write(R_s, v) \prec read(R_s)$ *e*
$\nexists\ write(R_s, u) \in r : write(R_s, v) \prec write(R_s, u) \prec read(R_s)$; *ou*

(ii) $\exists\ u \in \mathbb{D}(R_s), \exists\ write(R_s, u) \in r : write(R_s, u) \parallel read(R_s)$.

Registrador regular. Um registrador *regular* R_r é seguro (*safe*), no sentido de que uma leitura que não é concorrente a uma escrita sobre o mesmo registrador retorna o último valor que foi escrito. No entanto, oferece uma semântica mais forte, na medida em que uma leitura sobre um registrador regular que é concorrente a uma ou mais operações de escrita sobre o mesmo registrador pode obter ou o valor (um dos valores) que está sendo escrito ou o último valor escrito antes das operações de escrita concorrentes à leitura. Além disso, se uma escrita se sobrepõe a duas leituras sequenciais, a primeira leitura pode retornar o novo valor, que está sendo escrito; e a segunda retornar o valor antigo.

No cenário representado pela Figura 2.3, a operação $read_1(R_r)$, que não é concorrente a qualquer escrita, obtém o último valor escrito em R_r, isto é, x. A segunda leitura, $read_2(R_r)$, é concorrente às escritas $write_2(R_r, y)$ e $write_3(R_r, z)$ e, portanto, pode obter qualquer um dos valores que estão sendo escritos: y ou z; ou o valor escrito anteriormente: x. A terceira leitura, $read_3(R_r)$, é concorrente à escrita $write_3(R_r, z)$, então pode retornar o valor antigo: y, ou o que está sendo escrito: z.

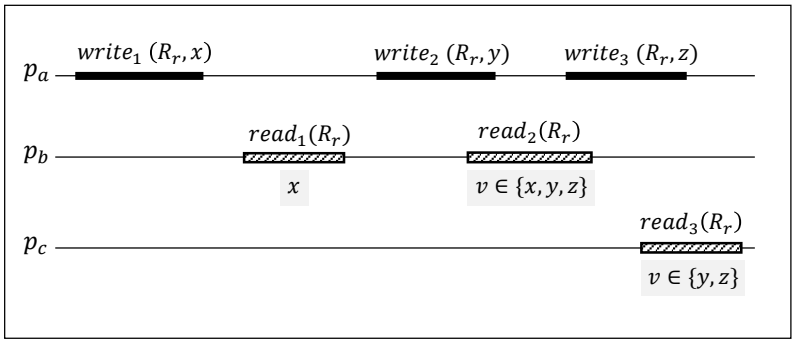

Figura 2.3 Comportamento de um registrador regular R_r. $read_1(R_r)$ retorna x. $read_2(R_r)$ pode retornar x, y ou z. $read_3(R_r)$ pode retornar y ou z. Nesse caso, pode acontecer uma inversão *novo/antigo*: $read_2(R_r)$ retornar z e $read_3(R_r)$ retornar y.

Pode acontecer, inclusive, de $read_3(R_r)$, que é iniciada após o término de $read_2(R_r)$, retornar o valor *antigo*, y, e $read_2(R_r)$ retornar o valor *novo*, z. Esse fenômeno, que é permitido para registradores regulares, é chamado de *inversão novo-antigo* (GUERRAOUI; RAYNAL, 2007; LAMPORT, 1986). Apesar disso, é possível escrever algoritmos corretos com registradores regulares, a exemplo dos algoritmos apresentados nos Capítulos 3 e 4. Os valores que podem ser retornados por uma operação de leitura num registrador regular são definidos a seguir.

Definição 2.4 (Valor admissível para um registrador *regular*: $read(R_r)$)**.** *Dado um registrador regular R_r, uma execução r e uma operação $read(R_r) \in r$, um valor $v \in \mathbb{D}(R_r)$ é admissível para $read(R_r)$ se:*

(i) $\exists\ write(R_r, v) \in r : write(R_r, v) \parallel read(R_r)$; *ou*

(ii) $\exists\ write(R_r, v) \in r : write(R_r, v) \prec read(R_r) \wedge$
$\nexists\ write'(R_r, v') \in r, v \neq v' : write(R_r, v) \prec write'(R_r, v') \prec read(R_r)$.

A seguinte proposição é verificada para registradores regulares.

Proposição 2.2.1. *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 invocação e resposta das operações executadas por p e q:

(1) tw_i^p e tw_r^p: invocação e resposta da operação $write^p(R_i, v)$;
tr_i^p e tr_r^p: invocação e resposta da operação $read^p(R_j)$.

Portanto: $tw_i^p < tw_r^p < tr_i^p < tr_r^p \Rightarrow tw_r^p < tr_i^p$ (A).

(2) tw_i^q e tw_r^q: invocação e resposta da operação $write^q(R_j, w)$;
tr_i^q e tr_r^q: invocação e resposta da operação $read^q(R_i)$.

Portanto: $tw_i^q < tw_r^q < tr_i^q < tr_r^q \Rightarrow tw_r^q < tr_i^q$ (B).

Da hipótese, p não obtém w. Isso só pode acontecer, no caso de registradores regulares, se $read^p(R_j)$ termina antes ou, possivelmente, é concorrente a $write^q(R_i, v)$. Portanto, $tr_i^p \leq tw_r^q$ (C). Por outro lado, ainda pela hipótese, q não obtém v. Então, $tr_i^q \leq tw_r^p$ (D). De (C), (B), (D) e (A) tem-se: $tr_i^p \leq tw_r^q < tr_i^q \leq tw_r^p < tr_i^p$, uma contradição. *Proposição*2.2.1□

Registrador atômico. Um registrador *atômico* (LAMPORT, 1986), R_a, é o que oferece a semântica mais forte. Ele é um registrador regular que não permite *inversão novo-antigo*. Assim, uma leitura que não é concorrente a uma escrita, sempre obtém o valor atual (último valor escrito). Uma leitura que se sobrepõe 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. Entretanto, se duas leituras sequenciais se sobrepõem a uma operação de escrita e a primeira leitura obtém o valor novo (o que está sendo correntemente escrito) então a leitura subsequente não pode obter o valor antigo, isto é, obrigatoriamente retorna o valor novo. Sendo assim, o valor admissível para uma operação de leitura é assim definido:

Definição 2.5 (Valor admissível para um registrador atômico: $read(R_a)$)**.** *Dado um registrador atômico R_a, uma execução r e uma operação $read(R_a) \in r$, um valor $v \in \mathbb{D}(R_a)$ é admissível para $read(R_a)$ se:*

(i) $\exists\ write(R_a, v) \in r : write(R_a, v) \parallel read(R_a)$; *ou*

(ii) $\exists\ write(R_a, v) \in r : write(R_a, v) \prec read(R_a) \wedge$
$\nexists\ write'(R_a, v') \in r, v \neq v' : write(R_a, v) \prec write'(R_a, v') \prec read(R_a) \wedge$
$\nexists\ write''(R_a, v''), read'(R_a)[v''] \in r, v \neq v'' :$
$write''(R_a, v'') \parallel read'(R_a)[v''] \prec read(R_a) \parallel write''(R_a, v'')$.

Na Figura 2.4, o retorno da operação $read_1(R_a)$, que não é concorrente a qualquer escrita, é o mesmo que seria para um registrador *safe* ou *regular*, isto é, o último valor escrito: x. A segunda leitura, $read_2(R_a)$, assim como no registrador regular, pode obter qualquer um dos valores que estão sendo escritos: y ou z; ou o valor antigo, escrito antes das operações de escrita concorrentes à leitura: x. A operação $read_3(R_a)$ poderia retornar o valor antigo y ou o que está sendo escrito: z. No entanto, para um registrador atômico existe uma restrição a mais. A leitura $read_3(R_a)$ só pode retornar o valor antigo, y, se $read_2(R_a)$ também retorna y ou x. Caso contrário, isto é, se $read_2(R_a)$ retorna o valor novo, z, então $read_3(R_a)$ também deve retornar o valor novo, garantindo que a precedência $read_2(R_a) \prec read_3(R_a)$ seja obedecida.

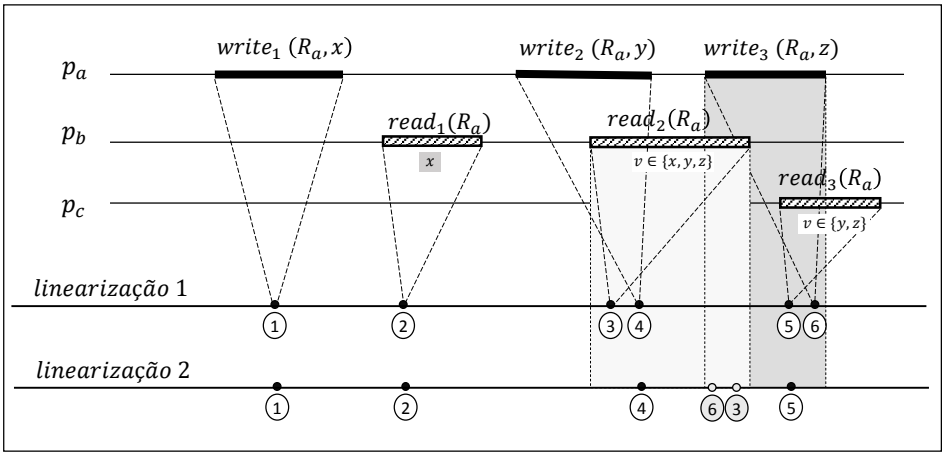

Figura 2.4 Comportamento de um registrador atômico R_a em uma execução concorrente e duas possíveis linearizações. Os pontos de linearização aparecem numerados de 1 a 6, conforme a execução sequencial correspondente. Na segunda opção, as operações $read_2(R_a)$ e $write_3(R_a, z)$ mudam de posição na execução sequencial.

A propriedade *atomicidade* corresponde à propriedade de consistência *linearizabilidade* (*linearizability*), introduzida por (HERLIHY; WING, 1990), para objetos concorrentes, isto é, um objeto que é compartilhado entre processos concorrentes. Intuitivamente, a propriedade provê uma ilusão de que cada operação em uma execução concorrente é executada atomicamente em algum instante entre sua invocação e retorno. Dessa forma, é possível construir uma execução sequencial a partir dos instantes de tempo da "suposta" execução atômica de cada operação.

Considere atribuir a cada operação da execução concorrente um instante de tempo qualquer, localizado entre a invocação e o retorno da operação. Cada um desses instantes de tempo é chamado *ponto de linearização* da operação correspondente. Assuma que cada uma dessas operações é executada atomicamente em seu ponto de linearização. Se a sequência formada pelos pontos de linearização de todas as operações é consistente com a especificação do objeto, diz-se que o objeto em questão é *linearizável*. A sequência assim obtida é chamada de *linearização* da execução concorrente.

Na Figura 2.4, duas possíveis linearizações são construídas a partir da execução concorrente. Na primeira, os pontos de linearização aparecem numerados, sequencialmente, de 1 a 6, resultando na seguinte execução sequencial: $\{write_1(R_a, x),\ read_1(R_a)[x],\ read_2(R_a)[x],\ write_2(R_a, y),\ read_3(R_a)[y],\ write_3(R_a, z)\}$. Note que cada operação de leitura retorna sempre o valor escrito na última operação de escrita precedente. Assim, $read_1(R_a)$ e $read_2(R_a)$ retornam x, enquanto que $read_3(R_a)$ retorna y.

Para obter a outra linearização, o ponto de linearização da operação $read_2(R_a)$ (número 3) é movido para a direita e o da operação $write_3(R_a, z)$ (número 6) para esquerda. A nova ordem das operações passa a ser 1-2-4-6-3-5, isto é, $\{write_1(R_a, x),\ read_1(R_a)[x],\ write_2(R_a, y),\ write_3(R_a, z),\ read_2(R_a)[z],\ read_3(R_a)[z]\}$. Nesse caso, a operação $read_2(R_a)$ aparece na execução linear depois da escrita $write_3(R_a, z)$ e portanto, deve retornar z. Consequentemente, a leitura $read_3(R_a)$ também será posterior à

escrita de z, obrigando esta leitura a retornar o mesmo valor.

Uma vez que um registrador atômico é um registrador regular com a restrição de que inversões novo-antigo não são permitidas, a Proposição 2.2.1 também é válida para registradores atômicos.

2.2.1.2 Objeto Conjunto. O outro objeto compartilhado a ser considerado é o *conjunto*, o qual armazena um conjunto de valores. No contexto da tese, um conjunto apresenta similaridades com os introduzidos em (DELPORTE-GALLET; FAUCONNIER, 2009) e (BALDONI; BONOMI; RAYNAL, 2010b)[2], e provê as seguintes operações primitivas:

(i) $insert(C, v)$, que insere o elemento v no conjunto C e retorna OK; e

(ii) $get(V)$, que retorna o conjunto de elementos constantes em C sem modificá-lo.

É assumido que um conjunto é inicialmente vazio. À semelhança do objeto registrador, pode-se definir alguns tipos de conjunto de acordo com o domínio de valores que podem ser armazenados, os direitos de acesso pelos processos e ainda conforme o comportamento do conjunto quando sob acesso concorrente.

Domínio de estados. Um conjunto é dito *binário* se seu domínio de estados é $\{0, 1\}$ e *multivalorado* se qualquer valor inteiro pode ser um elemento do conjunto.

Direitos de acesso. Considerando o acesso de processos, podem ser definidas as seguintes classes de conjuntos:

- 1-*escritor*-1-*leitor* ($1W1R$) – apenas um processo pode invocar operações $insert(C, v)$ e apenas um pode invocar operações $get(C)$;
- 1-*escritor-múltiplos-leitores* ($1WMR$) – apenas um processo pode invocar operações $insert(C, v)$ e múltiplos processos podem invocar $get(C)$;
- *múltiplos-escritores-múltiplos-leitores* ($MWMR$) – múltiplos processos podem invocar operações $insert(C, v)$ e $get(C)$.

Consistência. No contexto da tese, cada conjunto C, assim como um registrador, só pode ser modificado por um processo (escritor), de modo que as operações *insert* sobre um mesmo conjunto são sempre sequenciais e a semântica de um conjunto C é dada pelo valor retornado por uma operação $get(C)$. Um conjunto não é linearizável. Dessa maneira, uma $get(C)$ que não é concorrente a uma $insert(C, v)$ retorna o conjunto correntemente em C, i.e., todos os valores inseridos em C por operações $insert(C, -)$ precedentes, ou seu valor inicial, se não houver operação $insert(C, -)$ precedente. Contudo, se uma operação $get(C)$ ocorre concorrentemente a uma operação $insert(C, -)$ ou mais, o conjunto de valores retornado pode ser:

[2] O objeto *set* no artigo de (BALDONI; BONOMI; RAYNAL, 2010b) provê, além das operações $add(v)$ e $get()$, a operação $remove(v)$, que remove o elemento v do conjunto.

(i) o conjunto anteriormente armazenado em C, antes de qualquer $insert(C, -)$ concorrente; ou

(ii) o conjunto anteriormente armazenado em C acrescentado de qualquer valor v (ou mais de um deles) sendo inserido concorrentemente.

Os valores que podem ser retornados por uma operação $get(C)$ são definidos em função dos conjuntos C_{pre} e C_{con}. O conjunto C_{pre} contém todos os valores inseridos em C em operações precedentes a $get(C)$. O conjunto C_{con} contém todos os valores inseridos em C em operações concorrentes a $get(C)$.

Definição 2.6 (Conjunto precedente para uma operação $get(C)$)**.** *Dada uma execução r e uma operação $get(C) \in r$, um conjunto C_{pre} contém todos os valores $v \in \mathbb{D}(C)$ tais que:*

$$\exists\ insert(C, v) \in r : insert(C, v) \prec get(C).$$

Definição 2.7 (Conjunto concorrente para uma operação $get(C)$)**.** *Dada uma execução r e uma operação $get(C) \in r$, um conjunto C_{con} contém todos os valores $v \in \mathbb{D}(C)$ tais que:*

$$\exists\ insert(C, v) \in r : insert(C, v) \parallel get(C).$$

Definição 2.8 (Conjunto admissível para uma operação $get(C)$)**.** *Dada uma execução r, uma operação $get(C) \in r$, um conjunto precedente C_{pre} e um conjunto concorrente C_{con}; um conjunto C_{adm} é admissível para $get(C)$ se:*

(i) $C_{pre} \subseteq C_{adm}\ \wedge$

(ii) $(C_{adm} \setminus C_{pre}) \subseteq C_{con}$.

A Figura 2.5 representa uma execução concorrente sobre o conjunto C e os possíveis valores de retorno. A operação $get_1(C)$ retorna um conjunto com o único elemento que foi inserido em uma operação de inserção precedente a $get_1(C)$. Considerando a operação $get_2(C)$, concorrente a $insert_2(C, y)$ e $insert_3(C, z)$, obtém-se os seguintes conjuntos:

$$C_{pre} = \{x\};\ C_{con} = \{y, z\};\ C_{adm} = \{x\} \vee \{x, y\} \vee \{x, z\} \vee \{x, y, z\}.$$

Um conjunto admissível para $get_2(C)$ contém, obrigatoriamente, o conjunto precedente, $\{x\}$, e qualquer combinação possível entre os elementos de C_{con}.

2.3 MODELOS DE SINCRONISMO

O modelo de sincronismo define como são limitados os tempos para alguns eventos no sistema e para o desvio do relógio local de cada processo com relação ao tempo real. Considerando esses parâmetros, um sistema pode ser definido como *síncrono*, *assíncrono* ou ainda pode ter uma classificação intermediária.

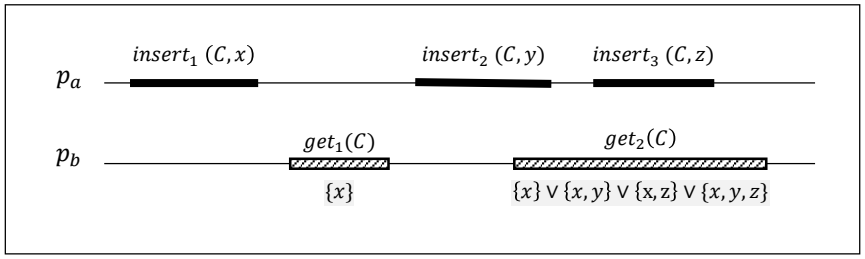

Figura 2.5 Valores admissíveis para um conjunto numa execução concorrente.

2.3.1 Sistema Distribuído Síncrono

Um sistema distribuído síncrono em que os processos se comunicam através de trocas de mensagens, apresenta as seguintes propriedades temporais (HADZILACOS; TOUEG, 1994; COULOURIS et al., 2011):

1. Existe um limite superior conhecido sobre o tempo que leva para qualquer processo dar um passo.

2. Existe um limite superior conhecido sobre o tempo que leva para o envio, transferência e recepção de uma mensagem sobre qualquer canal de comunicação.

Em um ambiente síncrono, onde a comunicação se dá por memória compartilhada, o tempo para as operações de acesso à memória também é limitado (GUERRAOUI; RAYNAL, 2006);

Uma vez que os limites de tempo são bem definidos neste modelo, é possível detectar a falha de um processo ou de um canal de transmissão quando os limites são violados. (HADZILACOS; TOUEG, 1994) provam que existe solução determinística para o problema do consenso em tal modelo de sistema.

2.3.2 Sistema Distribuído Assíncrono

Em um sistema completamente assíncrono não existem suposições sobre a velocidade relativa em que os processos executam operações locais nem sobre o tempo consumido com operações de comunicação. Isso significa que pode haver um intervalo de tempo arbitrariamente longo, embora finito, entre dois passos de um processo qualquer ou entre o início e o término de uma operação de leitura em um registrador.

Uma vez que no modelo assíncrono não existe um relógio global, os acessos à memória compartilhada, por diferentes processos, podem ser intercalados e sobrepostos de maneira arbitrária (NISHIMURA, 1990).

Como uma consequência direta do assincronismo, protocolos para sistemas com esta propriedade não podem utilizar temporizadores para controlar o progresso dos processos, tornando impossível distinguir um processo muito lento de um processo faltoso, o que leva, por exemplo, à impossibilidade do consenso na presença da falha de pelo menos um processo (FISCHER; LYNCH; PATERSON, 1985). Por outro lado, não existe sentido

em um sistema distribuído que não admita protocolos de consenso tolerantes a falhas. É por isso que muitos estudos têm se dedicado a propor propriedades adicionais no sentido de enriquecer tais sistemas com requisitos que sejam fortes o suficiente (em termos de sincronismo) para permitir o consenso e outras abstrações fundamentais, porém fracas o bastante para serem práticas.

(DOLEV; DWORK; STOCKMEYER, 1987) identificaram vários parâmetros relativos à sincronia e mostraram a resiliência dos protocolos de consenso desenhados para modelos que combinam os diversos parâmetros. (DWORK; LYNCH; STOCKMEYER, 1988) apresentam a noção de sistema *parcialmente síncrono*, que admite a existência de limites para a transferência de mensagens assim como para a velocidade relativa entre processos. Os autores apresentam duas versões para tais modelos de sistemas. Na primeira, os limites existem, porém são desconhecidos *a priori*. Na segunda, os limites existem e são conhecidos, mas só podem ser garantidos a partir de um instante que eles chamam de tempo de estabilização global (GST - *global stabilization time*). No mesmo artigo são apresentados protocolos de consenso para os modelos propostos.

(CRISTIAN; FETZER, 1999) admitem um limite de tempo para as transferências de mensagens assumindo que o sistema passa por períodos estáveis, em que falhas temporais não ocorrem, e períodos instáveis. Essa suposição dá origem ao que eles chamam de *modelo assíncrono temporizado* (*timed asychronous model*).

Via de regra, as propriedades adicionais propostas na literatura, consideram que todos ou parte dos processos participantes e/ou canais de comunicação funcionam de maneira síncrona (*timely*) a partir de um determinado instante e variam essencialmente no teor de sincronia e conectividade da rede subjacente (AGUILERA et al., 2004; HUTLE et al., 2009; FERNÁNDEZ et al., 2010). Outra abordagem, menos frequente, ao invés de assumir limites de tempo para processos ou transferências de mensagens, baseia-se num padrão de troca de mensagens que independe da definição de limites de tempo (MOSTEFAOUI et al., 2006; ARANTES et al., 2013).

Modelos estendidos com detectores de falhas não confiáveis (CHANDRA; TOUEG, 1996), são os modelos mais simples e mais gerais, no sentido de que os requisitos de sincronismo são encapsulados em mecanismos chamados *detectores de falhas*. Esta é a abordagem adotada neste trabalho e será vista em mais detalhes na Seção 2.6.

2.4 MODELOS DE FALHAS

Uma falha ocorre num sistema quando um de seus componentes desvia-se do comportamento correto, isto é, do comportamento especificado. Existem diversos tipos de falhas que podem afetar, de maneiras distintas, os componentes de um sistema. Por outro lado, as falhas em componentes podem afetar o funcionamento de serviços oferecidos pelo sistema. Um sistema tolerante a falhas deve continuar a fornecer seus serviços a despeito das falhas de componentes.

A implementação de um serviço tolerante a falhas depende do modelo de falhas considerado, isto é, da forma como os componentes falhos se desviam do comportamento esperado. Nesta seção, são tratados os modelos de falhas dos componentes de interesse: processos, e objetos compartilhados.

2.4.1 Processos

Uma falha de processo ocorre quando ele se desvia do comportamento esperado. As principais falhas que afetam os processos em um sistema distribuído são: *falhas por parada* (*crash*), *falhas por omissão*, *falhas arbitrárias* e *falhas de temporização*. A Tabela 2.1 apresenta um resumo desses tipos de falhas conforme descritos na literatura (CRISTIAN, 1991; CACHIN; GUERRAOUI; RODRIGUES, 2011).

No contexto desta tese, considera-se que os processos podem falhar *por parada*, o que ficará explícito na apresentação do modelo de sistema particular considerado no desenvolvimento de cada algoritmo (Capítulos 3, 4 e 5). Um processo que falha por parada executa corretamente seu algoritmo até que repentinamente para de dar passos. Isso significa que ele para de executar qualquer computação local ou operação sobre os objetos compartilhados e depois que para, não se recupera mais. Tal processo é dito *faltoso*. Um processo que não falha é dito *correto* e permanece executando pra sempre, isto é, executa infinitos passos de seu algoritmo. "Pra sempre"aqui significa "enquanto durar determinada execução do algoritmo". Por exemplo, para um protocolo de consenso que depende da implementação de um detector de falhas, é suficiente e necessário que a implementação do detector continue executando corretamente até que o protocolo de consenso convirja.

Tipo de Falha	**Comportamento do Processo**
Omissão	O processo deixa de responder a alguma entrada (*input*). Em sistemas de passagem de mensagens, destacam-se ainda as falhas: **omissão de envio** – o processo completa o envio de uma mensagem, mas ela não é inserida no *buffer* de saída; e **omissão de recepção** – uma mensagem é inserida no *buffer* de entrada do processo, mas ele não a recebe efetivamente.
Parada (*crash* ou *crash-stop*)	O processo para prematuramente de dar passos e não se recupera mais.
Parada com Recuperação (*crash-recovery*)	O processo mantém-se parando (*crashing*) e recuperando-se pra sempre. Neste modelo, um processo que para e se recupera um número finito de vezes é considerado correto.
Arbitrária ou **Bizantina**	O processo se desvia arbitrariamente de seu algoritmo, isto é, exibe um comportamento qualquer distinto do programado. Por exemplo, pode enviar mensagens arbitrárias, realizar uma operação que não deveria, parar, etc.
Temporização	**Falha de Relógio**. Existe um intervalo de tempo no qual o desvio do relógio do processo com relação ao tempo real excede o limite estabelecido. **Falha de Desempenho**. O processo completa um passo em um tempo maior ou menor que o respectivo limite especificado.

Tabela 2.1 Modelos de falhas de processos.

Normalmente, quando se projeta o algoritmo de alguma abstração para um sistema distribuído, tem-se em vista que o algoritmo funcionará corretamente desde que um limite superior para o número de processos que podem vir a falhar, f, seja obedecido. Diz-se então que tal algoritmo é *f-resiliente*. Um algoritmo é *livre de espera* (*wait-free*) (HERLIHY, 1991), se é garantido que qualquer processo correto completa o algoritmo em um número finito de (seus próprios) passos, qualquer seja o comportamento dos demais processos. Portanto, o processo termina sua computação independente do número de processos que falhem por parada. Algoritmos *wait-free* são $(n-1)$-*resilientes* quanto a falhas por parada de processos, não são sujeitos a impasse (*dead-lock*) e não sofrem degradação devido a preempção por escalonamento, falta de página ou falta de cache. Em contrapartida, normalmente são complexos e consomem muita memória (MERRITT; TAUBENFELD, 2013).

As falhas por parada, por omissão e arbitrárias afetam sistemas assíncronos e síncronos, enquanto que as falhas de temporização só dizem respeito a sistemas síncronos, uma vez que são relativas à extrapolação dos limites de tempo, os quais só são presentes nestes sistemas. As falhas por parada, por omissão e de temporização não implicam em um desvio, por parte dos processos, de seu algoritmo, isto é, a sequência de passos executada por um processo é sempre consistente com o autômato associado ao processo, seja ele faltoso ou correto. Por isso falhas destes tipos são consideradas *benignas*. Estas falhas são mais comuns em sistemas distribuídos, e um modelo de sistema em que os processos se comunicam através de memória compartilhada onde até $n-1$ processos podem falhar por parada é fundamental (HERLIHY; RAJSBAUM; RAYNAL, 2013).

Classe de Falha	**Modelo de Falha**	**Comportamento do Objeto**
Responsiva	Parada (*crash* ou *fail-stop*)	O objeto comporta-se corretamente até falhar. Quando falha, retorna $\perp$ para qualquer operação.
	Omissão	O objeto faltoso pode retornar uma resposta correta a uma invocação de um processo p e retornar $\perp$ à invocação subsequente de um processo q.
	Arbitrária	O objeto faltoso pode retornar respostas arbitrárias.
Não-Responsiva	Parada (*crash* ou *fail-silent*)	O objeto comporta-se corretamente até falhar. Quando falha, para de responder.
	Omissão	O objeto faltoso deixa de responder à invocação de um subconjunto arbitrário de processos mas responde à de outros.
	Arbitrária	O objeto faltoso pode não responder a algumas invocações ou retornar respostas arbitrárias.

Tabela 2.2 Modelos de falhas de objetos compartilhados.

2.4.2 Memória Compartilhada

De um modo mais geral, objetos compartilhados podem falhar de maneira *responsiva* ou *não responsiva*. Em cada uma dessas classes de falhas, três modelos são identificados por (JAYANTI; CHANDRA; TOUEG, 1992), conforme mostra a Tabela 2.2.

O foco desta tese está sobre falhas por parada. No entanto, sabe-se que é possível implementar registradores confiáveis a partir de registradores base sujeitos a falhas por parada, sejam elas responsivas ou não (HERLIHY; RAJSBAUM; RAYNAL, 2013). Então, uma vez que é possível contornar as falhas de objetos, considera-se aqui o modelo mais simples, em que os registradores são *confiáveis*. Isto é, sempre retornam corretamente às invocações e o comportamento regular ou atômico (conforme definido no modelo particular para cada protocolo proposto) é garantido.

2.5 CONSENSO

Uma abstração fundamental em sistemas distribuídos tolerantes a falhas é o acordo. Em muitas situações, os participantes corretos precisam concordar com relação a uma determinada informação, para manter a integridade do sistema. Por exemplo, em sistemas que envolvem replicação de dados, os processos precisam concordar quanto à ordem das mensagens que contém operações sobre os dados. Outra necessidade comum é a de que os processos concordem a respeito da lista de componentes faltosos.

Via de regra, para se desenvolver aplicações confiáveis em sistemas distribuídos, é preciso que um ou mais problemas de acordo sejam resolvidos. Dentre os problemas de acordo, o *consenso* é o mais importante e é o foco desta tese. Na realidade, o consenso pode ser considerado como um subproblema comum e está no coração dos problemas de acordo.

A importância do problema do consenso se justifica, portanto, devido à sua onipresença e por isso mesmo tem sido alvo de estudos teóricos e práticos nas últimas décadas.

2.5.1 Definição do Problema

A ideia por trás do consenso é compartilhar informação entre um grupo de processos. Em particular, é essencial que o consenso possa ser atingido mesmo em presença de falhas. Isto é, ainda que alguns processos falhem, os processos corretos devem estar aptos a tomar uma decisão comum. Para isso, cada processo p_i propõe um valor inicial, v_i, e todos os processos corretos devem decidir um mesmo valor v, entre os valores propostos.

O problema do consenso então, é definido em termos das seguintes propriedades (CHANDRA; TOUEG, 1996). As duas primeiras dizem respeito ao aspecto da segurança (*safety*), enquanto que a terceira refere-se à vivacidade (*liveness*).

Propriedade 2.5.1 (Validade)**.** *Se um processo decide um valor v, então v é o valor inicial de algum processo.*

Propriedade 2.5.2 (Acordo)**.** *Se o valor v é a decisão de algum processo, então todo processo correto que decide, decide pelo mesmo valor v.*

Propriedade 2.5.3 (Terminação)**.** *Todo processo correto em algum momento acaba por decidir algum valor.*

Na versão tradicional do consenso, exige-se o acordo apenas entre os processos corretos. Na versão *uniforme*, uma propriedade mais forte é considerada para o acordo, exigindo que todos os processos que cheguem a uma decisão, corretos ou faltosos, decidam pelo mesmo valor:

Propriedade 2.5.4 (Acordo Uniforme)**.** *Se o valor v é a decisão de algum processo, então todo processo que decide, decide pelo mesmo valor v.*

Usando a terminologia de memória compartilhada, conveniente ao propósito da tese, um protocolo de consenso é definido como segue. Cada processo p_i possui um registrador compartilhado de entrada, $input_i$, inicializado com o valor inicial de p_i, $v_i \in \mathbb{D}$, e um registrador de saída, $decision_i$, com domínio de estados igual a $\mathbb{D} \cup \{\perp\}$, inicializado com $\perp$, representando um valor que não pode ser proposto por qualquer processo. Os estados em que o registrador $decision_i$ tem um valor distinto de $\perp$ são estados de *decisão*.

Uma execução do consenso é iniciada quando cada p_i propõe seu valor inicial e termina quando p_i escreve em seu registrador $decision_i$ o valor v decidido por todos os processos. Um estado de decisão deve ser consistente com as propriedades 2.5.1, 2.5.2 (2.5.4) e 2.5.3.

2.5.2 Impossibilidade do Consenso

Apesar do enunciado simples, o problema do consenso não possui solução determinística em sistemas assíncronos sujeitos a falhas de processos por parada, onde os processos se comunicam através de passagem de mensagens (FISCHER; LYNCH; PATERSON, 1985), ou por memória compartilhada (LOUI; ABU-AMARA, 1987). Intuitivamente, esse resultado se explica pela impossibilidade de distinguir, dada a ausência de limites de tempo, um processo faltoso de um processo muito lento.

Assim, com o intuito de contornar a impossibilidade, algumas abordagens têm sido propostas, no sentido de enriquecer o sistema com requisitos de sincronismo que sejam suficientes para viabilizar a solução do problema. Uma abordagem introduzida no protocolo Paxos (LAMPORT, 1998) e que vem sendo aplicada em muitos algoritmos de acordo é separar as propriedades de segurança (*safety*) das propriedades de vivacidade (*liveness*).

As propriedades de segurança determinam que alguma coisa ruim não aconteça, por isso não devem ser violadas nunca. Por exemplo, num algoritmo de consenso, dois processos não podem decidir dois valores distintos. Já as propriedades de vivacidade, definem que alguma coisa boa deve acontecer, ainda que sejam postergadas por períodos arbitrários. No caso do consenso, uma decisão pode ser adiada, contanto que se garanta que ela acabará acontecendo em algum momento.

Uma abstração que se adapta bem à abordagem descrita acima é introduzida por Chandra e Toueg: *detectores de falhas não confiáveis* (CHANDRA; TOUEG, 1996). Um detector de falhas encapsula requisitos de sincronismo, permitindo que os protocolos de interesse contem com a ajuda desses mecanismos para detectar falhas de processos por parada. Esta é a abordagem adotada neste trabalho e será vista em mais detalhes na próxima seção.

2.6 DETECTORES DE FALHAS

Um detector de falhas não confiável (CHANDRA; TOUEG, 1996) é um mecanismo distribuído com um módulo associado a cada processo e que funciona como um oráculo. Cada módulo provê o processo ao qual é vinculado com dicas a respeito de falhas de processos. "Não confiável" significa que um detector pode cometer erros de duas maneiras: pode suspeitar de processos corretos e deixar de suspeitar de processos faltosos. Além disso, enquanto um módulo considera um processo p correto, outro pode considerá-lo suspeito e se em um instante t um módulo considera p faltoso, no instante $t + 1$ pode considerá-lo correto.

O grau de (não) confiabilidade de um detector é definido pelas propriedades *completude* e *acurácia*. A primeira diz respeito à capacidade de detectar corretamente os processos que de fato falham. Nos protocolos que se baseiam em detectores de falhas, essa propriedade é utilizada para impedir que os processos esperem indefinidamente por um processo que falhou. A propriedade acurácia refere-se aos erros cometidos em considerar suspeitos processos que na realidade são corretos (falso-positivos). Ela é usada para garantir que ao cabo de um tempo, processos corretos (possivelmente, pelo menos um processo) não sejam equivocadamente considerados suspeitos, postergando indefinidamente a terminação do protocolo.

Chandra e Toueg (1996) definem dois níveis para a propriedade *completude* e quatro níveis para a propriedade *acurácia*, os quais são descritos a seguir. Combinando cada propriedade de completude com cada uma de acurácia, eles definem oito classes de detectores de falhas.

Completude

Completude forte (*Strong completeness*). Ao cabo de um tempo, todo processo que falha é permanentemente considerado suspeito por todo processo correto.

Completude fraca (*Weak completeness*). Ao cabo de um tempo, todo processo que falha é permanentemente considerado suspeito por algum processo correto.

Acurácia

Acurácia forte (*Strong accuracy*). Nenhum processo é considerado suspeito antes que falhe.

Acurácia fraca (*Weak accuracy*). Algum processo correto nunca é considerado suspeito.

Acurácia forte após um tempo (*Eventual strong accuracy*). Existe um tempo após o qual nenhum processo correto é considerado suspeito por qualquer processo correto.

Acurácia fraca após um tempo (*Eventual weak accuracy*). Existe um tempo após o qual algum processo correto nunca é considerado suspeito por qualquer processo correto.

O projeto de algoritmos baseados em detectores de falhas é assim simplificado, uma vez que é possível abstrair a ocorrência de falhas através do detector e concentrar-se nas propriedades de segurança do algoritmo. Como os detectores são definidos em termos de propriedades abstratas, os algoritmos são projetados e suas provas elaboradas com base nas propriedades do detector. Além disso, os requisitos inerentes à detecção de

falhas não precisam ser atendidos pelo algoritmo, que delega esta tarefa ao detector cuja implementação é independente do algoritmo em questão.

Entre as classes de detectores de falhas originalmente propostas por Chandra e Toueg (1996), a classe $\diamondsuit\mathcal{S}$ é particularmente interessante por que é minimal no sentido de que é a classe mais fraca, em termos das informações sobre falhas que fornecem (ou do sincronismo que encapsulam), que permitem a realização do consenso através de algoritmos determinísticos (CHANDRA; HADZILACOS; TOUEG, 1996; DELPORTE-GALLET et al., 2004).

Outra classe de oráculos bastante atrativa é a de *detectores de líder* Ω (Ômega), que é equivalente, em termos do sincronismo encapsulado, à classe $\diamondsuit\mathcal{S}$ (DELPORTE-GALLET et al., 2004; CHANDRA; HADZILACOS; TOUEG, 1996), quando o conjunto de participantes é conhecido.

2.6.1 Detectores de Falhas $\diamondsuit\mathcal{S}$

Cada módulo de um oráculo $\diamondsuit\mathcal{S}$, quando requisitado pelo processo p_i vinculado a ele, retorna uma lista de processos, $supected_i$, que o oráculo considera suspeitos. Esses módulos podem cometer erros, durante um período *mau*, incluindo em suas listas de suspeitos, processos corretos ou deixando de incluir processos faltosos. Mas a classe $\diamondsuit\mathcal{S}$ satisfaz algumas propriedades que garantem que a partir de um determinado instante t, um período *bom* é iniciado em que o oráculo é capaz de fornecer informações confiáveis o suficiente para fazer convergir, por exemplo, um algoritmo de consenso. Um detector $\diamondsuit\mathcal{S}$ satisfaz às seguintes propriedades:

Propriedade 2.6.1 (Completude forte (*strong completeness*))**.** *Ao cabo de um tempo, todo processo que falha será considerado permanentemente suspeito por todo processo correto.*

Propriedade 2.6.2 (Acurácia fraca após um tempo (*eventual weak accuracy*))**.** *Existe um instante a partir do qual algum processo correto jamais será considerado suspeito por qualquer processo correto.*

A primeira propriedade garante que a partir de determinado instante t, a lista de suspeitos fornecida para cada processo conterá todos os processos que falharam, impedindo que algum processo correto confie em um processo faltoso. A segunda se encarrega de que, a partir de algum instante t', pelo menos um processo correto jamais seja, equivocadamente, considerado suspeito. Ainda que esses instantes não sejam conhecidos, a simples existência deles permite garantir o término dos algoritmos de consenso baseados em detectores $\diamondsuit\mathcal{S}$.

2.6.2 Detectores de Líder Ω

Um *detector de líder após um tempo*, conhecido como Ω, provê uma funcionalidade de *eleição de líder após um tempo*. O módulo local do detector num processo p_i lhe provê uma função *leader*() que retorna a identidade de um processo que ele considera correto naquele instante – o *líder*. Diz-se então que o processo p_i *confia* naquele líder.

Ao cabo de um tempo, um único líder correto é eleito por todos os processos corretos, mas eles não têm conhecimento do instante em que isso ocorre. Dessa maneira, durante um período arbitrário de tempo (mau), vários líderes podem coexistir, mas a propriedade de liderança após um tempo, satisfeita pelos oráculos da classe líder, garante que a partir de um instante t, toda invocação de $leader()$ retorna a identidade l do mesmo processo.

Propriedade 2.6.3 (Liderança após um tempo (*eventual leadership*))**.** *Existe um processo correto p_l e um instante t, após o qual o detector fornece a identidade de p_l (i.e., o mesmo líder) para todos os processos corretos.*

Algoritmos de consenso baseados em um detector Ω podem ser encontrados em (LAMPORT, 1986; MOSTEFAOUI; RAYNAL, 2000; DUTTA; GUERRAOUI, 2002; LAMPORT, 1998).

2.6.3 Protocolos Indulgentes em Relação aos Detectores

As propriedades que definem a qualidade das informações fornecidas pelos oráculos estabelecem comportamentos que devem ser verificados apenas a partir de um instante arbitrário, mas em caráter permanente. Por exemplo, um detector da classe $\diamondsuit\mathcal{S}$ satisfaz à propriedade *completude forte*, de modo que *a partir de um instante t*, toda vez que qualquer processo correto p_i invoca o seu oráculo, recebe uma lista $supected_i$ com todos os processos que falharam. Antes do instante t, nada é garantido.

Da mesma forma, um oráculo $\diamondsuit\mathcal{S}$ atende à propriedade *acurácia fraca após um tempo*, que garante que a partir de um instante t, pelo menos um processo correto, p_x, não será considerado suspeito por qualquer processo correto. Então, $\forall t' \geq t$, $\forall p_i \in \Pi$, $p_x \notin suspected_i$, . Mais uma vez, não há garantias sobre o comportamento do oráculo antes de t. O mesmo ocorre com a classe Ω, que só garante um comportamento consistente dos módulos do detector a partir de um certo instante t em diante.

Como se pode depreender das propriedades dos oráculos $\diamondsuit\mathcal{S}$ e Ω, esses detectores oferecem garantias para um período de tempo considerado *bom*, ou *estável* ("a partir de um instante t"), mas seu comportamento pode ser completamente arbitrário durante períodos *maus* ou *instáveis* ou *anárquicos*. Isto é, em períodos maus, tudo é permitido: falsas suspeições, ou deixar de suspeitar de processos faltosos. Até mesmo situações conflitantes, como por exemplo, um oráculo local Ω retornar a identidade de um processo p_x, no mesmo instante em que outro módulo local retorna p_y.

Apesar desse caráter não-confiável, detectores $\diamondsuit\mathcal{S}$ e Ω possuem uma ótima qualidade. Eles permitem a implementação de protocolos *indulgentes* (GUERRAOUI, 2000), isto é, qualquer que seja o comportamento do oráculo subjacente (desde que respeitadas suas especificações), as propriedades de segurança (*safety*) de um protocolo baseado num desses oráculos são sempre atendidas.

O pior que pode acontecer durante os períodos maus, se informações conflitantes sobre falhas de processos transitarem pelo sistema, é que o protocolo não produza saídas. Por exemplo, na execução de um algoritmo de consenso, nenhuma decisão é tomada. Mas deve-se observar que isso não é peremptório; mesmo em períodos instáveis, saídas podem ser produzidas e se isso ocorre, elas são corretas. Além disso, dado que as especificações

do detector sejam devidamente atendidas, no período bom, é garantido que o protocolo produza as saídas esperadas.

Na realidade, detectores de falhas são usados tão somente para garantir a terminação do protocolo, nada tendo a ver com as propriedades de segurança. Assim, desde que um detector satisfaça suas especificações, os protocolos que dele dependem, terminam, produzindo uma saída correta.

2.7 SISTEMAS DINÂMICOS

A maioria dos trabalhos sobre o projeto de algoritmos para memória compartilhada considera um modelo de sistema em que o conjunto de processos participantes é conhecido previamente. No entanto, o surgimento de uma nova classe de sistemas executando sobre uma infraestrutura como redes de sensores, redes P2P não estruturadas, grades oportunistas ou de computação em nuvem tem chamado a atenção para uma configuração diferenciada do conjunto de participantes, a qual pode ser constantemente alterada.

Um primeiro atributo, e mais evidente, é o número de participantes, que num modelo estático é fixo e conhecido, geralmente denotado n. Num sistema dinâmico, a cardinalidade do conjunto de participantes, Π, é variável e corresponde, em cada instante de tempo t, ao número de processos que entraram no sistema até t e não saíram ou falharam até então. Esse número de participantes pode ser limitado ou ilimitado, mas, via de regra, não é conhecido, a priori, pelos processos. Como consequência, os protocolos para esses sistemas não podem levar em conta o valor de n.

Outro parâmetro igualmente importante, considerado no modelo, é o número máximo de processos faltosos que se admite, denotado f. O projeto de protocolos para sistemas distribuídos estáticos geralmente conta com o conhecimento de n e de f e oferecem garantias de corretude desde que o número de falhas não ultrapasse o limite f. No modelo dinâmico, entretanto, nem sempre esse parâmetro está disponível.

Devido às constantes alterações na configuração do conjunto de participantes, os processos num sistema dinâmico estão sujeitos a um conhecimento restrito desse conjunto. É possível que cada processo só tenha consciência de um subconjunto limitado de processos com os quais pode interagir, a saber, sua *vizinhança* (BALDONI et al., 2007). Assim, o *conhecimento* a respeito do conjunto de participantes é outro atributo que difere o modelo dinâmico do estático.

Para modelar o dinamismo em sistemas distribuídos, várias abordagens têm sido propostas (GAFNI; MERRITT; TAUBENFELD, 2001; AGUILERA, 2004; MOSTEFAOUI et al., 2005; BALDONI et al., 2007). De modo geral, os modelos se distinguem por um *padrão de chegada* de processos – *finito* ou *infinito*; e pela *concorrência*, definida pelo número de processos que entra menos o número de processos que deixa o sistema a cada instante. A concorrência também pode ser limitada por um valor finito conhecido – *k-limitada* (*k-bounded*), limitada por um valor finito desconhecido – *n-limitada* (*n-bounded*) ou *ilimitada* (*unbounded*). Do ponto de vista prático, o número de processos ativos é sempre limitado.

A movimentação de entrada e saída de processos em instantes arbitrários, bem como a ocorrência de falhas (que de certo modo corresponde à saída de processos, embora

involuntária) é denominada *churn* (*agitação*) (GODFREY; SHENKER; STOICA, 2006). Esse fenômeno introduz um fator adicional de incerteza nos sistemas distribuídos na medida em que um processo não pode mais ter uma visão precisa do número total de componentes no sistema. Como consequência, as abstrações destinadas a esses ambientes e suas implementações devem considerar o *churn* como mais um parâmetro no modelo do sistema e o projeto de algoritmos deve levar em conta suposições a seu respeito.

É nesse sentido que diversos trabalhos, a maioria deles de caráter experimental, têm proposto modelos para o *churn*, os quais se dividem, basicamente, em duas categorias: Modelos *baseados nos nós* e modelos *baseados no sistema*. Os primeiros consideram distribuições de probabilidade sobre os instantes em que os processos estão ativos ou inativos visando especificar um padrão de comportamento sobre processos (e.g. (GODFREY; SHENKER; STOICA, 2006; KONG; BRIDGEWATER; ROYCHOWDHURY, 2006; BLOND; FESSANT; MERRER, 2009)). Os modelos baseados no sistema buscam um padrão para o sistema como um todo. Visam especificar a quantidade de processos que se movimentam bem como os instantes de movimentação, modelando esse comportamento através de um padrão de escalonamento ((MOSTEFAOUI et al., 2005; KO; HOQUE; GUPTA, 2008; BALDONI et al., 2009)).

Na realidade, qualquer modelo baseado em sistema é equivalente a algum modelo baseado em nó. No entanto, modelos baseados em sistema são, via de regra, mais gerais, permitindo a análise de uma classe de propriedades de estabilidade para protocolos distribuídos arbitrários. Essas propriedades, chamadas *quiescentes* (KO; HOQUE; GUPTA, 2008), são necessárias para garantir a terminação dos protocolos. Uma propriedade quiescente é uma propriedade global do sistema que pode ter valor verdadeiro ou falso em qualquer instante da execução de um protocolo. Uma vez que ela assuma um valor verdadeiro, esse valor se mantém até que ocorra algum *churn*.

Uma propriedade de terminação de um protocolo que pode tornar-se falsa por causa do *churn* e em seguida ser convertida em verdadeira devido às ações individuais dos processos executando o protocolo é uma propriedade quiescente. Por exemplo, no problema de eleição de líder, uma propriedade quiescente é a de que um único líder permaneça vivo e seja conhecido por todos os processos vivos. É desejável, portanto, que essa propriedade mantenha-se verdadeira por um tempo "suficiente" (KO; HOQUE; GUPTA, 2008).

A intuição por trás da necessidade de uma propriedade quiescente é a exigência de alguma estabilidade no sistema para que alguma computação seja completada. Felizmente, apesar das constantes alterações em sua configuração, sistemas dinâmicos são caracterizados por períodos instáveis seguidos de períodos estáveis. Na verdade, a terminação é possível, mesmo na presença de *churn*, mas nenhuma garantia é fornecida sem que um período estável ocorra. Portanto, é preciso que no modelo de sistema considerado alguma condição de estabilidade seja satisfeita por um período de tempo "suficiente" (MOSTEFAOUI et al., 2005).

Seguindo essa linha de raciocínio, Mostefaoui et al (2005) propõem um modelo de sistema dinâmico que considera um parâmetro α que representa o número mínimo de processos que devem estar ativos durante o período em que a computação é executada. Por exemplo, a execução de um oráculo que fornece uma identidade de líder para um protocolo de consenso numa camada superior, deve contar que pelo menos α processos

permaneçam ativos garantindo a estabilidade da eleição até que o consenso seja alcançado.

Seja t_b o instante de tempo em que a execução da aplicação na camada superior se inicia e t_e o instante em que ela termina. Então a condição quiescente exigida para o protocolo do oráculo é que durante o intervalo $I = [t_b, t_e]$, o número de processos ativos durante I seja maior ou igual a α. (MOSTEFAOUI et al., 2005) argumentam que essa suposição combina com a observação experimental num sistema P2P (GUMMADI et al., 2003) de que um pequeno número de pares persiste por um longo tempo.

O problema de encontrar as suposições mais fracas que um sistema dinâmico assíncrono deve satisfazer para que abstrações como o consenso ou Ω possam ser implementadas ainda está em aberto. Mas deve-se notar que pelo menos um processo deve permanecer ativo por um tempo suficiente no sistema para que alguma computação seja possível (LARREA et al., 2012).

2.7.1 Memória compartilhada implementada sobre sistemas dinâmicos

Como foi dito na Seção 2.2.1, um sistema de memória compartilhada pode ser implementado por uma camada inferior de um sistema distribuído. Na realidade, qualquer sistema distribuído deve prover os processos com serviços fundamentais tais como a implementação de objetos compartilhados. Assim como acontece com qualquer abstração implementada sobre um sistema dinâmico, uma memória distribuída compartilhada deve levar em consideração o efeito do *churn* nesse ambiente.

Uma abordagem interessante, introduzida em (BALDONI; BONOMI; RAYNAL, 2012), considera que a implementação da memória compartilhada pelo sistema subjacente se encarrega de cuidar do dinamismo. Eles fornecem a implementação de um registrador regular para um modelo de sistema dinâmico conforme mostra a Figura 2.6. Um processo entra no sistema distribuído requisitando uma operação *join_system*, a qual permite que os processos que entram possam conectar-se àqueles que já estão no sistema. Um processo sai do sistema quando falha (*por parada*) ou requisitando a operação *leave_system*. É suposto que um protocolo gerencia as entradas e saídas de processos assim como a conectividade entre eles.

Uma vez que um processo tenha se ligado ao sistema dinâmico, pode decidir, de maneira autônoma, entrar na computação distribuída, a saber, a implementação do registrador compartilhado. De maneira análoga à camada inferior, um processo p_i entra na computação executando uma operação *join_computation*. O processo então é considerado ativo desde o instante do término desta operação até que requisite a operação *leave_computation* ou até que falhe. Enquanto *join_computation* está em andamento, p_i fica no modo "escutar", podendo receber e processar as mensagens enviadas pelos demais participantes da computação, isto é, aqueles que compartilham o registrador. Uma vez que a operação tenha sido completada, p_i terá obtido o valor atualizado do registrador e poderá requisitar as operações *read* e *write* sobre ele.

De modo semelhante à proposta de (BALDONI; BONOMI; RAYNAL, 2012), a abordagem adotada neste trabalho considera a invocação de uma operação $join()$ que se encarrega de criar registradores devidamente inicializados e prover os processos que chegam com um estado consistente dos objetos compartilhados. A implementação da operação

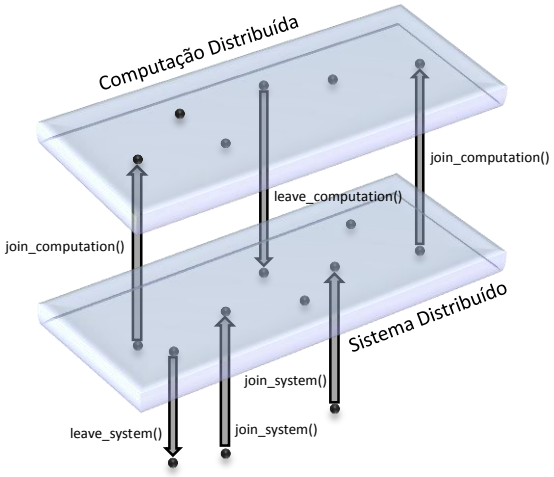

Figura 2.6 Implementação de uma memória compartilhada distribuída.
Fonte: (BALDONI; BONOMI; RAYNAL, 2012)

$join()$, entretanto, está fora do escopo da tese.

2.8 MEDIDAS DE COMPLEXIDADE

Existem várias medidas de complexidade utilizadas para algoritmos em memória compartilhada, as quais podem ser relativas ao tempo; ao trabalho executado por processo ou pela totalidade dos processos; aos acessos à memória compartilhada; ou ao espaço de memória utilizado (ASPNES, 2014). As seguintes métricas são consideradas nesta tese.

Complexidade de tempo do processo ou complexidade de passo. Esta métrica considera um limite de tempo para um passo de processo. Geralmente, o tempo de cada passo interno do processo é desprezado e assume-se que um processo demora uma unidade de tempo para acessar a memória compartilhada. O total de acessos à memória será então a complexidade de tempo do algoritmo. Uma análise mais detalhada pode considerar em separado cada tipo de operação. Por exemplo, o número máximo de leituras e de escritas que um processo pode ter que executar sobre registradores compartilhados para completar sua operação (MERRITT; TAUBENFELD, 2013; ASPNES, 2014).

Complexidade de espaço. Tal métrica diz respeito ao número e tamanho dos objetos compartilhados utilizados no algoritmo (MERRITT; TAUBENFELD, 2013).

Considerando, especificamente, algoritmos indulgentes, outras medidas de complexidade de tempo também são de interesse. Essas medidas são tomadas em alguns cenários específicos. Por exemplo, em execuções livres de falhas. De modo geral, elas são definidas no contexto de sistemas de passagem de mensagens (DUTTA; GUERRAOUI, 2002;

GUERRAOUI; RAYNAL, 2004) mas podem ser adaptadas ao modelo de memória compartilhada. Nesse caso, considera-se o número de acessos à memória compartilhada ao invés de passos de comunicação.

Eficiência relativa ao oráculo (Oracle-efficiency) Em algumas situações, é interessante considerar o comportamento do algoritmo em uma execução livre de falhas quando o oráculo é *perfeito*. Um oráculo é dito *perfeito* se logo no início da execução ele satisfaz as propriedades "após um tempo" (*eventual*). Por exemplo, um oráculo Ω é perfeito se no início da execução ele provê os processos com a identidade do mesmo processo correto. Um oráculo $\diamondsuit\mathcal{S}$ é dito perfeito se ele nunca suspeita de um processo correto e sempre suspeita de todo processo que falha.

Na realidade, a maioria das execuções de um sistema distribuído são "boas" (estáveis), isto é, as falhas são raras e os detectores, geralmente, não suspeitam de processos corretos (DUTTA; GUERRAOUI, 2002). Portanto, é importante otimizar os algoritmos para tais execuções.

Um parâmetro utilizado para aferir a eficiência de algoritmos de memória compartilhada refere-se às operações de escrita que permanecem sendo realizadas sobre a memória. Assim, a *eficiência quanto à escrita* (*write optimality*) de um algoritmo considera o número de processos que devem escrever para sempre na memória. Um algoritmo é dito *ótimo quanto à escrita* se apenas um processo permanece escrevendo pra sempre na memória compartilhada (GUERRAOUI; RAYNAL, 2006).

2.9 CONSIDERAÇÕES

Este capítulo apresentou a maioria dos conceitos e definições que serão utilizados ao longo da tese. À medida que forem sendo utilizadas nos próximos capítulos, as informações aqui presentes podem ser relembradas localmente ou a seção correspondente neste capítulo será referenciada. Alguns conceitos que dizem respeito, especificamente, a assuntos tratados em um capítulo em particular, serão apresentados quando necessário.

Capítulo

3

Este capítulo apresenta uma solução para o problema do consenso em um sistema dinâmico com memória compartilhada. Ao contrário do ambiente clássico, onde o conjunto de participantes e suas identidades são conhecidas por todos os processos, nas redes dinâmicas esse conhecimento global não está disponível. A estratégia utilizada na solução proposta é ampliar o conhecimento sobre os participantes do sistema com a ajuda de um detector de participantes e em seguida utilizar um algoritmo baseado no detector de líder para resolver o consenso.

CONSENSO COM PARTICIPANTES DESCONHECIDOS EM MEMÓRIA COMPARTILHADA

3.1 INTRODUÇÃO

TESTE DE FONTE TEXTSC

Sistemas distribuídos modernos, baseados em coleções *ad-hoc* de dispositivos de computação distribuída, redes P2P não-estruturadas, grades oportunistas ou nuvens, possibilitam a seus participantes acessarem serviços e informações, independentemente de sua localização, padrão de mobilidade ou topologia. Isso é possível através de um desenho de sistema altamente dinâmico que dispensa a existência de uma autoridade administrativa centralizada ou uma infraestrutura estática. Questões relacionadas ao desenho de soluções confiáveis que lidam com a natureza de alto dinamismo e auto-organização de tais redes são um campo muito ativo de pesquisa.

Por outro lado, os *problemas de acordo* são fundamentais em sistemas distribuídos confiáveis nos quais não existe uma autoridade central. O acordo permite que um conjunto de pares possa alcançar uma decisão unificada sobre alguns estados do sistema. Entre estes problemas, o mais importante é o *consenso*. No problema do consenso, cada processo propõe um valor e todos os processos corretos devem atingir uma decisão comum que consiste em escolher um dentre os valores propostos. É bem estabelecido, entretanto, que não existem algoritmos determinísticos para resolver o problema do consenso em sistemas assíncronos sujeitos a falhas por parada (*crash*) (FISCHER; LYNCH; PATERSON, 1985).

A maioria dos protocolos de consenso na literatura baseia-se no paradigma de passagem de mensagens para comunicação entre processos (CHANDRA; TOUEG, 1996;

GREVE; TIXEUIL, 2007). Entretanto, considerando os enormes avanços no desenvolvimento de máquinas multinúcleo, onde vários processos compartilham uma memória física comum; e as classes de sistemas implementadas sobre redes modernas, como os serviços centrados em dados altamente disponíveis e tolerantes a falhas, ressalta-se a importância do fornecimento de serviços fundamentais para sistemas de memória compartilhada.

Entre esses sistemas, destacam-se as redes de área de armazenamento – SANs, nas quais conjuntos de discos – NADs – podem ser acessados diretamente por qualquer processo. Essas redes são definidas como sistema dinâmicos, uma vez que não é necessário o conhecimento prévio sobre o conjunto de processos que participam do sistema. Como os NADs aceitam requisições de leitura e escrita em blocos de dados, esses blocos podem ser modelados como registradores compartilhados.

É por isso que alguns trabalhos têm se dedicado a propor soluções de serviços básicos considerando o modelo de memória compartilhada. Em especial, o consenso baseado em discos (GAFNI; LAMPORT, 2003; CHOCKLER; MALKHI, 2002; AGUILERA; ENGLERT; GAFNI, 2003; GUERRAOUI; RAYNAL, 2007) que são capazes de fornecer serviços de armazenamento distribuído confiáveis.

Resolver o consenso em sistemas de memória compartilhada sujeitos a falhas por parada de processos foi previamente investigado em (DELPORTE-GALLET; FAUCONNIER, 2009; LO; HADZILACOS, 1994; GUERRAOUI; RAYNAL, 2007) e descobriu-se que as condições necessárias e suficientes nem sempre coincidem com as dos sistemas de passagem de mensagem. Em particular, em sistemas estáticos, onde o conjunto de participantes é conhecido, enquanto a maioria dos processos corretos é necessária para resolver o consenso no modelo de passagem de mensagens, as soluções de memória compartilhada geralmente podem ser livres de espera (*wait-free*).

A busca por uma solução para o consenso que seja capaz de tolerar falhas por parada de processo e que assuma a comunicação por memória compartilhada é um dos objetivos desta tese. Especificamente, busca-se condições e protocolos capazes de resolver o consenso quando o conjunto de participantes e sua cardinalidade são desconhecidos. Este último critério corresponde aos requisitos dos sistemas atuais, quando um conjunto desconhecido e potencialmente enorme de processos colaboram para manter a consistência.

3.1.1 Do ponto de vista teórico

Embora o problema do consenso possa ser enunciado de forma simples, é bem sabido que não existe solução determinística para o consenso em sistemas distribuídos assíncronos sujeitos a falhas, mesmo que o número de processos faltosos seja limitado a um (FISCHER; LYNCH; PATERSON, 1985). Uma das maneiras de contornar essa impossibilidade é através da utilização de detectores de falhas, os quais são oráculos distribuídos que fornecem dicas sobre falhas de processos (CHANDRA; TOUEG, 1996) (ver Seção 2.6). Embora não sejam confiáveis, no sentido de que podem cometer um número arbitrário de erros, eles não comprometem as propriedades de corretude dos protocolos que os utilizam.

No contexto de sistemas distribuídos clássicos (isto é, onde cada processo tem o conhecimento de todos os outros no sistema), têm sido propostos vários algoritmos de resolução de consenso, incluindo os baseados em tais oráculos. Na verdade, algoritmos de

consenso baseados em detectores de falhas para memória compartilhada são a base para resolver o consenso em discos compartilhados (AGUILERA; ENGLERT; GAFNI, 2003; GAFNI; LAMPORT, 2003).

Recentemente, vários autores investigaram a possibilidade de resolver o consenso em sistemas distribuídos dinâmicos, por exemplo, (CAVIN; SASSON; SCHIPER, 2004, 2005; GREVE; TIXEUIL, 2007, 2010; ALCHIERI et al., 2008), mas para o modelo de passagem de mensagens. Em particular, (CAVIN; SASSON; SCHIPER, 2005; GREVE; TIXEUIL, 2007, 2010) investigaram a possibilidade de resolver o *consenso tolerante a falhas com participantes desconhecidos* (ou FT-CUP). FT-CUP mantém as mesmas propriedades do problema do consenso original, mas supõe que os nós não têm consciência sobre o conjunto de processos no sistema, Π, ou até mesmo sua cardinalidade, n.

No entanto, se alguma cooperação global é esperada, é preciso que os processos tenham algum conhecimento parcial sobre os outros. Por essa razão, esses estudos propõem a abstração *detector de participantes* - PD. Um detector de participantes também é um oráculo distribuído, mas que fornece dicas sobre os processos que participam da computação.

A informação fornecida pelo PD pode ser representada por um *grafo da conectividade do conhecimento*, G_{di}, que é um grafo direcionado cujos vértices são os nós que participam no sistema. Uma aresta (p_i, p_j) existe em G_{di} se p_i "conhece" p_j (ver Seção 3.4.3 para mais detalhes). Com base no grafo da conectividade do conhecimento inicial, pode-se definir condições de conectividade do conhecimento necessárias e suficientes para resolver FT-CUP em ambientes assíncronos. Os detectores de participantes são classificados de acordo com as características do grafo da conectividade do conhecimento que eles geram. A classe k-OSR (Definição 3.4), por exemplo, reúne as condições minimais para resolver FT-CUP em um ambiente assíncrono sujeito a falhas por parada (GREVE; TIXEUIL, 2010).

Em teoria, mover-se de um modelo de passagem de mensagens para um modelo de memória compartilhada deve permitir aumentar o número de falhas toleradas (por exemplo, de $n/2$ para $n-1$, quando o grafo da conectividade do conhecimento é fortemente conectado). No entanto, nenhum transformador de modelo algorítmico disponível (ATTIYA; BAR-NOY; DOLEV, 1995; ATTIYA, 2000; GUERRAOUI; LEVY, 2004; ALON et al., 2011; DOLEV et al., 2012) retêm essa propriedade (no máximo $n/2$ falhas são normalmente obtidas na transformação para memória compartilhada) e também exigem conhecimento completo sobre os outros processos (isto é, o conjunto Π é conhecido por todos os processos), quebrando os requisitos de soluções baseadas em detector de participantes (CAVIN; SASSON; SCHIPER, 2005; GREVE; TIXEUIL, 2007, 2010). Essas duas deficiências apontam para o projeto de uma solução dedicada.

Dessa forma, é introduzido um novo conjunto de algoritmos *wait-free* para resolver o consenso uniforme com base em um detector de falhas da classe Ω, em um sistema assíncrono sujeito a falhas, com participantes desconhecidos, onde a comunicação entre os processos ocorre via memória compartilhada. Os protocolos propostos anteriormente por (GREVE; TIXEUIL, 2007) para o modelo de passagem de mensagens são revisitados e otimizados (em termos de simplicidade) para memória compartilhada.

Até onde se sabe, esta é a primeira proposta que resolve FT-CUP usando memória

compartilhada. Mais especificamente, são apresentados aqui:

(i) uma definição do problema FT-CUP para o modelo de memória compartilhada com uma nova definição para o detector de participantes k-OSR;

(ii) algoritmos para este modelo que são tão eficientes (complexidade de espaço/tempo) quanto os de (GREVE; TIXEUIL, 2007);

(iii) prova de que um detector de participantes k-OSR, juntamente com um detector de líder Ω fornecem as condições suficientes para chegar a um consenso neste modelo.

3.1.2 Trabalhos Relacionados

Até onde se sabe, os trabalhos de Cavin *et al.* (2005) e de Greve e Tixeuil (2007, 2010) são os únicos que propõem as condições sobre conectividade do conhecimento para resolver consenso em uma rede sujeita a falhas por parada com participantes desconhecidos, mas para o modelo de passagem de mensagem.

Cavin *et al* (2005) identificaram o detector de falhas perfeito ($\mathcal{P}$) como um requisito de sincronia necessário para resolver FT-CUP quando o grafo inicial da conectividade do conhecimento tem requisitos de conectividade mínimos. Mas $\mathcal{P}$ só pode ser efetivamente implementado em sistemas síncronos. Resolver FT-CUP em um cenário com a conectividade do conhecimento mais fraca possível, requer condições de sincronia mais fortes. Por sua vez, sincronia forte não se encaixa com o alto dinamismo, a descentralização total e auto-organização das redes de sensores, *ad hoc* e SANs. Além disso, mesmo com um detector de falhas perfeito, quando se considera a conectividade do conhecimento mínimo, a versão uniforme do FT-CUP (onde *todos* os processos que decidem devem decidir o mesmo valor) não pode ser resolvida em redes desconhecidas.

Greve e Tixeuil (2007) apresentaram a proposta dual para resolver FT-CUP. Eles consideram os requisitos mínimos de *sincronia* já identificados para resolver o consenso no modelo clássico (conhecimento completo), e buscaram os requisitos mínimos para o *grafo da conectividade do conhecimento* que permita resolver o problema. O requisito mínimo de sincronia é denotado pelo detector de falhas $\diamondsuit\mathcal{S}$, equivalente ao detector de líder Ω (CHANDRA; TOUEG, 1996). A desvantagem notável da proposta é que deve existir um componente central (o poço em (GREVE; TIXEUIL, 2007)) no qual a maioria dos processos seja correta.

Alchieri *et al.* (2008) estudaram o problema do consenso tolerante a falhas bizantinas com participantes desconhecidos em sistemas de passagem de mensagens, a saber, BFT-CUP. Usando um conjunto de protocolos e reduções de modo semelhante a (GREVE; TIXEUIL, 2007), eles proveem as condições suficientes e necessárias para resolver BFT-CUP em redes desconhecidas com a restrição adicional de que os participantes do sistema podem se comportar de forma maliciosa.

Delporte-Gallet e Fauconnier (2009) apresentam algoritmos de consenso em um modelo híbrido, com participantes desconhecidos e anônimos, considerando-se que as interações ocorrem por registradores atômicos compartilhados, ao contrário da proposta aqui que considera objetos não atômicos. O foco de seu trabalho é em sistemas anônimos,

onde, ao contrário do modelo aqui considerado, os processos não têm identidade. Eles propõem soluções para sistemas eventualmente síncronos (a partir de algum instante, todos os processos se comunicam em tempo ou um mesmo processo se comunica em tempo com os outros). No entanto, seu modelo não considera a abstração detector de participantes nem as condições sobre o grafo da conectividade do conhecimento. Além disso, como mencionado na Seção 2.2.1, registradores atômicos são mais fortes que regulares, de modo que sua implementação é mais complexa.

Até onde se sabe, não foram propostas soluções para FT-CUP em um sistema assíncrono, com participantes desconhecidos não anônimos, que se comunicam via memória compartilhada - SM FT-CUP. Neste sentido, o presente trabalho é pioneiro. A Tabela 3.1 resume os resultados conhecidos considerando CUP, FT-CUP e BFT-CUP. Ao que se sabe, esses são os únicos trabalhos que estudaram as condições de conectividade do conhecimento aptas a resolver o consenso em um sistema com participantes desconhecidos. Com exceção deste, todos consideram o modelo de passagem de mensagem.

Abordagem	Modelo de Falha	Comunicação	Detector de Participantes	Conectividade	Modelo de Sincronia
CUP (CAVIN; SASSON; SCHIPER, 2004)	sem falhas	passagem de mensagem	OSR	OSR	assíncrono
FT-CUP (CAVIN; SASSON; SCHIPER, 2005)	parada	passagem de mensagem	OSR	OSR + padrão de falhas seguro	assíncrono $+ \mathcal{P}$
FT-CUP (GREVE; TIXEUIL, 2007, 2010)	parada	passagem de mensagem	k-OSR	$f+1$ caminhos disjuntos nos nós	assíncrono $+ \Omega$
BFT-CUP (ALCHIERI et al., 2008)	bizantino	passagem de mensagem	k-OSR	$2f+1$caminhos disjuntos nos nós	mesmo do protocolo de consenso
SM FT-CUP (este trabalho)	parada	memória compartilhada	k-OSR	$f+1$ caminhos disjuntos nos nós	assíncrono $+ \Omega$

Tabela 3.1 Soluções para Consenso com Participantes Desconhecidos

O restante deste capítulo é estruturado como segue: o modelo do sistema considerado é apresentado na Seção 3.2. O problema do consenso é reapresentado rapidamente na Seção 3.3. A Seção 3.4 apresenta alguns conceitos que não foram vistos no Capítulo 2. A Seção 3.5 introduz uma ideia geral sobre os algoritmos que são apresentados nas Seções 3.6, 3.7 e 3.8. A Seção 3.11 conclui o capítulo com algumas considerações.

3.2 MODELO DO SISTEMA

Considera-se o modelo de chegada infinita com concorrência limitada (GAFNI; MERRITT; TAUBENFELD, 2001). O sistema distribuído é composto por um conjunto infinito de processos, mas a cada execução, apenas um conjunto finito de n processos, $\Pi = \{p_1, p_2, ..., p_n\}$, participa. Ao contrário do que ocorre no modelo de sistema clássico, em que todo processo conhece Π, a rede é *desconhecida*, o que significa que um processo p_i pode estar consciente de apenas um subconjunto dos processos $\pi_i \subseteq \Pi$. π_i é um conjunto arbitrário, definido de acordo com a abstração *detector de participantes* (ver Seção 3.4.1). Além disso, n também é desconhecido dos processos e pode variar de execução a execução.

Um processo pode falhar por parada e, após parar, ele não se recupera, sendo considerado *faltoso*. Um processo que nunca falha, comporta-se de acordo com sua especificação e é dito *correto*. O número máximo de processos que pode falhar, f, é conhecido por todos os processos. Em um passo do algoritmo, um processo executa atomicamente uma das seguintes operações:

(i) obtém o valor em um objeto compartilhado;

(ii) altera o valor de um objeto compartilhado;

(iii) consulta um oráculo: detector de participantes ou de falhas;

(iv) executa uma operação interna;

(v) executa a operação $return()$.

Os processos se comunicam através de *objetos compartilhados*. Nos algoritmos apresentados neste capítulo são utilizados dois tipos de objetos compartilhados, definidos conforme a Seção 2.2.1: *registrador regular* e *conjunto*. Não são feitas quaisquer suposições sobre a velocidade relativa dos processos, seja para dar um passo interno ou para acessar a memória compartilhada, isto é, o sistema é assíncrono. Considera-se a existência de um relógio discreto global fictício, cuja faixa de tiques, $\mathcal{T}$, é o conjunto dos números naturais. Este relógio não é conhecidos pelos processos e só é utilizado para estabelecer algumas definições ou provar propriedades dos protocolos.

3.2.1 Registradores Regulares

Considera-se um arranjo de registradores regulares confiáveis, multivalorados, R, do tipo 1-*escritor*-M-*leitores* (ou $1WMR$). Nos algoritmos propostos, o registrador $R[i]$ está associado ao processo p_i, seu único escritor. Um outro processo p_j pode ler $R[i]$ se e somente se $p_i \in \pi_j$. Cada registrador possui três campos que serão descritos posteriormente.

3.2.2 Objetos Conjunto

O sistema dispõe também de um arranjo de objetos conjunto confiáveis, $Known$, que podem armazenar valores inteiros. Um processo p_i é o único que pode inserir elementos em $Known[i]$ e um processo p_j pode obter o valor do conjunto $Known[i]$ se e só se $p_i \in \pi_j$.

3.2.3 Detector de líder Ômega (Ω)

O sistema é estendido com um detector de líder da classe Ω (ver Seção 2.6.2). Um detector Ω satisfaz a propriedade *liderança após um tempo*, que garante que após um tempo, todo processo correto p_i que invocar a função $leader_i()$ recebe a identidade do mesmo processo correto, l.

3.3 CONSENSO

O problema do consenso consiste em obter um valor de decisão único para todos os processos que seja igual ao valor proposto por algum dos participantes. Em sua versão uniforme, o problema é definido, formalmente, pelas propriedades *Validade*, *Acordo uniforme* e *Terminação* (ver Seção 2.5).

O protocolo apresentado neste capítulo resolve SM FT-CUP, o consenso tolerante a falhas com participantes desconhecidos em memória compartilhada, onde até f processos podem falhar por parada, e também sua versão uniforme: SM FT-CUP Uniforme.

3.4 CONDIÇÕES DE CONECTIVIDADE PARA O CONSENSO EM SISTEMAS SUJEITOS A FALHAS

A maioria dos trabalhos disponíveis em consenso considera que o conjunto de participantes Π é conhecido por todos os processos no sistema. Em redes de sensores, redes móveis *ad hoc*, grades oportunistas, ou redes P2P não-estruturadas, essa suposição é claramente irrealista já que os processos podem ser mantidos por diferentes autoridades administrativas, têm muitos instantes de acordar, inicializações, etc. Obviamente, algum conhecimento sobre outros nós é necessário para executar qualquer algoritmo distribuído não trivial. Por exemplo, o envio de mensagens "Olá" para a vizinhança poderia ser uma forma possível de cada processo obter algum conhecimento sobre os outros processos.

3.4.1 Detector de Participantes: Uma Abstração da Conectividade do Conhecimento

A noção de detector de participantes (PD) foi proposta por (CAVIN; SASSON; SCHIPER, 2004) para prover os processos com um conhecimento inicial sobre os membros do sistema. Eles podem ser vistos como oráculos distribuídos fornecendo informação sobre quais processos participam do sistema. Denota-se $i.\mathsf{PD}$ o módulo do detector de participantes no processo p_i. Quando requisitado por p_i, $i.\mathsf{PD}$ retorna um subconjunto de processos em Π. Esta informação pode evoluir entre requisições.

Definição 3.1 (Detector de Participantes)**.** *O detector de participantes $i.\mathsf{PD}$ do processo p_i retorna um conjunto $\pi_i \subseteq \Pi$. Seja $i.\mathsf{PD}(t)$ a consulta de p_i no instante $t \in \mathcal{T}$. Essa consulta deve satisfazer as seguintes propriedades:*

> **Propriedade 3.4.1** (Acurácia da Informação)**.** *O detector de participantes não comete erros:*
> $$\forall p_i \in \Pi, \forall t \in \mathcal{T} : i.\mathsf{PD}(t) \subseteq \Pi.$$
>
> **Propriedade 3.4.2** (Inclusão da Informação)**.** *A informação retornada pelo detector de participantes é não decrescente no decorrer do tempo:*
> $$\forall p_i \in \Pi, \forall t, t' \in \mathcal{T}, t' \geq t : i.\mathsf{PD}(t) \subseteq i.\mathsf{PD}(t').$$

Para simplificar a notação, diz-se que $p_j \in i.\mathsf{PD}$ quando $p_j \in i.\mathsf{PD}(t)$ para algum t. É importante notar que, ao contrário dos detectores de falhas, as informações fornecidas

pelos detectores de participantes não lidam com falhas. Assim, o subconjunto de processos retornado por i.PD pode conter processos corretos ou faltosos.

3.4.2 Notação de Grafos

Diz-se que um grafo direcionado $D(V, E)$ é *k-fortemente* conectado se, para qualquer par de nós (v_i, v_j), v_i pode alcançar v_j por k caminhos disjuntos nos nós. D é *fortemente conectado* se k =1.

No grafo acíclico direcionado (DAG), obtido pela redução de G_{di} a suas componentes fortemente conectadas, um *poço* é uma componente da qual não saem arestas. Pelo teorema de Menger (YELLEN; GROSS, 1998), sabe-se que o número mínimo de nós cuja remoção de $G_{di}(V, E)$ desconecta o nó v_i de v_j, $\forall v_i, v_j \in V$ é o número maximal de caminhos disjuntos nos nós de v_i a v_j.

Observação 3.4.1. Se um grafo é *k-fortemente* conectado, a remoção de $(k-1)$ nós deixa pelo menos um caminho entre qualquer par de nós (v_i, v_j).

3.4.3 O Grafo da Conectividade do Conhecimento

A relação de conhecimento fornecida pelos detectores de participantes pode ser modelada como um grafo da conectividade do conhecimento do sistema, G_{di}. Este grafo é direcionado, uma vez que o conhecimento que é fornecido pelo detector de participantes não é necessariamente bidirecional (ou seja, é possível que $p_j \in i$.PD e $p_i \notin j$.PD). A Figura 3.1 mostra um grafo G_{di} induzido por um PD. Na figura, o DAG obtido pela redução de G_{di} a suas componentes fortemente conectadas tem três componentes, entre elas, um poço.

Adicionalmente, como o conhecimento sobre os participantes evolui com o tempo, G_{di} evolui com o tempo também. Seja $G_{di}(t) = (V, E_{di}(t))$ o grafo direcionado obtido da saída do PD no instante $t \in \mathcal{T}$. Então, $V = \Pi$ e $(p_i, p_j) \in E_{di}(t)$ *se e somente se* $p_j \in i$.PD(t), i.e., p_i conhece p_j no instante t.

A propriedade Inclusão da Informação é válida para G_{di} de modo que $\forall t' \geq t : G_{di}(t) \subseteq G_{di}(t')$, o que significa que novos relacionamentos de conhecimento entre processos podem ser identificados com o passar do tempo mas não descartados.

Analogamente, $G(t) = (V, E(t))$ é o grafo *não direcionado* obtido a partir de PD em t, isto é, $V = \Pi$ e $(p_i, p_j) \in E(t)$ *se e somente se* $p_j \in i$.PD(t) ou $p_i \in j$.PD(t), assim, $\forall t' \geq t : G(t) \subseteq G(t')$.

Na abordagem seguida neste trabalho, não é necessário precisar o instante de tempo em que G_{di} é composto. Assim, considera-se as seguintes definições:

Definição 3.2 (Grafo Não Direcionado da Conectividade do Conhecimento: $G = (V, E)$)**.** *$G = G(t_i) \supseteq G(t_{i-1}) \supseteq ... \supseteq G(t_0)$ é a sequência de subgrafos não direcionados geradores de G, tal que $V = \Pi, E(t_i) \supseteq E(t_{i-1}), \forall t_i \geq t_{i-1}$.*

Definição 3.3 (Grafo da Conectividade do Conhecimento: $G_{di} = (V, E_{di})$)**.**

$G_{di} = G_{di}(t_i) \supseteq G_{di}(t_{i-1}) \supseteq ... \supseteq G_{di}(t_0)$ é a sequência de subgrafos geradores de G_{di}, tal que $V = \Pi, E_{di}(t_i) \supseteq E_{di}(t_{i-1}), \forall t_i \geq t_{i-1}$.

3.4.4 Classes de Detectores de Participantes

Com base nas propriedades do grafo da conectividade do conhecimento, G_{di}, gerado pelo PD, várias classes de detectores de participantes podem ser obtidas , (CAVIN; SASSON; SCHIPER, 2004; GREVE; TIXEUIL, 2007, 2010). A classe que reúne as condições minimais para resolver FT-CUP em um ambiente assíncrono é a k-OSR. Essa classe foi recentemente revista em (GREVE; TIXEUIL, 2010) e será usada neste trabalho.

Definição 3.4 (Detector de participantes k-One Sink Reducibility: PD k-OSR)**.** *A classe de detectores de participantes k-OSR contém todos os detectores que fornecem informação sobre o conhecimento do conjunto de participantes do sistema que induzem um grafo direcionado G_{di} tal que:*

(1) O grafo não direcionado da conectividade do conhecimento, G, obtido de G_{di} é conectado;

(2) O grafo acíclico direcionado (DAG) obtido pela redução de G_{di} a suas componentes fortemente conectadas tem exatamente um poço, a saber, G_{sink};

(3) G_{sink} é k-fortemente conectada;

(4) $\forall p_i, p_j \in V$, tal que p_i não está em G_{sink}, p_j está em G_{sink}, existem k-caminhos disjuntos nos nós de p_i a p_j.

Deve-se observar que é possível resolver o consenso com a ajuda de um detector de participantes k-OSR desde que $f < k$. Além disso, a componente poço pode ser constituída de apenas um nó, desde que este seja correto.

A Figura 3.1 mostra um grafo G_{di} induzido por um PD 2-OSR. Note-se que existe apenas uma componente poço, a qual é 2-fortemente conectada. Além disso, todas as outras componentes são fortemente conectadas e existem pelo menos 2 caminhos de todo $p_i \notin G_{sink}$ para todo $p_j \in G_{sink}$. A notação $G_{di} \in k$-OSR significa que G_{di} é induzido por um PD da classe k-OSR, isto é, um PD que satisfaz as propriedades da Definição 3.4.

3.5 CONJUNTO DE ALGORITMOS PARA RESOLVER SM FT-CUP EM UM SISTEMA DE MEMÓRIA COMPARTILHADA

Nesta seção, é apresentado um conjunto de algoritmos capazes de resolver FT-CUP (consenso com participantes desconhecidos tolerante a falhas) no modelo de memória compartilhada – SM FT-CUP. Um aspecto importante da abordagem seguida nesse trabalho e em todos os outros que usam a abstração detector de participantes para resolver FT-CUP (CAVIN; SASSON; SCHIPER, 2004, 2005; GREVE; TIXEUIL, 2007, 2010; ALCHIERI et al., 2008), é a suposição de que $G_{di} \in k$-OSR desde o início da execução do sistema. É por isso que nos algoritmos apresentados, i.PD é requisitado *exatamente*

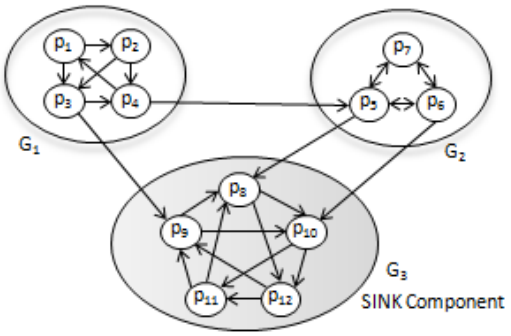

Figura 3.1 Grafo da Conectividade do Conhecimento Induzido por um PD 2-OSR – componentes fortemente conectadas.

uma vez no início da execução do sistema. De um ponto de vista prático, isto significa que o conjunto inicial de processos conhecidos por cada p_i é único, de modo que haverá apenas um grafo G_{di} representando o sistema.

3.5.1 O Raciocínio por trás dos Algoritmos

No início da execução, para obter uma visão inicial da composição do sistema, cada processo p_i primeiro consulta seu detector de participantes i.PD. O detector de participantes é consultado *exatamente uma vez* por cada p_i, de modo que a união dos conjuntos retornados por todos os i.PD é representada por um grafo único, isto é, o mesmo grafo da conectividade do conhecimento inicial, $G_{di} = (V, E_{di})$, é percebido por todos os nós no sistema. Seja $G_{sink} = (V_{sink}, E_{sink})$ a componente poço de G_{di}. Uma vez que o modelo assume que o PD pertence à classe k-OSR, tem-se que G_{di} satisfaz as propriedades de k-OSR (Definição 3.4).

Na solução proposta, uma sequência de algoritmos é executada por cada p_i com os seguintes objetivos:

1. Algoritmo COLLECT(): Expande o conhecimento sobre os participantes do sistema para além das informações retornadas pelo i.PD;

2. Algoritmo SINK(): Identifica a posição de p_i em G_{di}, isto é, se p_i está na componente poço, G_{sink}, ou não.

3. Algoritmo CONSENSUS(): Viabiliza a realização do consenso. Uma vez que G_{sink} é k-fortemente conectada, os nós nesta componente executam um algoritmo de consenso propriamente dito para decidir um valor v. Os demais nós então, obtém v e decidem o mesmo valor. Isto é possível porque embora os nós em G_{sink} não possam alcançar os nós nas outras componentes, todos os outros nós de G_{di} podem alcançar os nós em G_{sink};

No Capítulo 4, o conjunto de algoritmos é ampliado com a inclusão do algoritmo GENERICON para a realização do consenso propriamente dito.

3.5.2 Objetos Compartilhados e Notações

Os processos interagem através de um arranjo de registradores compartilhados, R, e um arranjo de conjuntos compartilhados, $Known$. Um conjunto $Known[i]$ armazena as identidades de processos conhecidos por p_i, i.e., $Known[i]$:$\{p_j | p_j \in \pi_i\}$. Inicialmente, $Known[i] = \emptyset$.

Cada registrador $R[i]$ possui três campos:

- $R[i].end\_pd$: um bit representando um valor booleano que indica se todas as identidades de processos retornadas pelo i.PD já estão em $Known[i]$. Inicialmente, $R[i].end\_pd = 0$ (falso).

- $R[i].end\_col$: um bit representando um valor booleano indicando se p_i terminou o algoritmo COLLECT(). Assim, o valor inicial de $R[i].end\_col$ é 0; quando p_i tiver completado a composição do conjunto de processos conhecidos, ele ajusta $R[i].end\_col$ para 1.

- $R[i].decision$: armazena o valor decidido por p_i. Este campo é inicializado com $\perp$, um valor padrão que não pode ser decidido por qualquer processo.

As operações sobre os objetos compartilhados são denotadas da seguinte forma.

- $write(R[i], v)$ escreve o valor v no registrador $R[i]$.

- $read(R[i], v)$ retorna o valor do registrador $R[i]$, e o armazena na variável local v.

- $insert(Known[i], v)$ insere o valor v no conjunto $Known[i]$.

- $get(Known[i], k)$ retorna o conjunto em $Known[i]$ e o armazena na variável local k.

Para facilitar a apresentação dos algoritmos, permite-se que um processo requisite uma leitura ou escreva em apenas um dos campos do registrador. Por exemplo, se a intenção é alterar apenas o campo *decision* de $R[i]$, escrevendo o valor v nele, pode-se usar a sentença $write(R[i].decision, v)$, para denotar a operação: $write(R[i].\langle end\_pd, end\_col, decision\rangle$, $R[i].\langle end\_pd, end\_col, v\rangle)$.

Quanto ao objeto conjunto, a função $Insert(S,\ S')$ denota a inserção sucessiva de cada elemento v do conjunto S' em S (possivelmente com a ajuda das operações $get(S')$ e $insert(S, v)$), ou seja, $S \leftarrow S \cup S'$. Considerando que cada processo p_i possui uma cópia local do seu conjunto compartilhado, permite-se que cada p_i acesse seu conjunto para leitura diretamente, sem fazer uso da operação $get()$.

3.6 O ALGORITMO COLLECT: AMPLIANDO O CONHECIMENTO INICIAL SOBRE OS PROCESSOS

A informação retornada pelo detector de participantes para p_i (denotado i.PD) pode não ser suficiente para prover os processos com o conhecimento necessário para atingir um

consenso diretamente. Na realidade, isso depende das propriedades do detector de participantes, ou, em outras palavras, das propriedades do grafo G_{di} formado. Por exemplo, se a informação dada resulta em um grafo G_{di} completo, então, é suficiente. Neste caso, o problema recai no modelo distribuído clássico em que todos os processos se conhecem, e qualquer protocolo clássico de consenso pode ser executado diretamente pelos processos.

Se, caso contrário, $G_{di} \in k\text{-}OSR$, mas não é completo, a informação dada inicialmente pelo PD não será suficiente. De qualquer forma, uma vez que o número de participantes é desconhecido e o sistema é assíncrono, um conjunto de protocolos deve ser executado antes de iniciar o protocolo de consenso em si.

Dessa forma, é proposto o Algoritmo 1 (COLLECT()). Ele estende o conhecimento de p_i sobre os participantes do sistema, explorando o grafo da conectividade do conhecimento resultante da consulta ao detector de participantes. COLLECT() tolera $f < k$ falhas e é completamente assíncrono, o que significa que não faz uso de qualquer oráculo detector de falhas para garantir progresso.

3.6.1 O Algoritmo

O processo p_i questiona iterativamente os processos recém-conhecidos por ele em G_{di} para melhorar o seu conhecimento sobre a rede até que nenhum conhecimento adicional possa ser adquirido. Assim, p_i realiza uma busca em G_{di} para identificar processos alcançáveis. Os processos cooperam compartilhando informações a fim de melhorar seu conhecimento com as visões de outros processos. O algoritmo contém dois laços que cuidam dessa cooperação. No final, sempre que PD $\in k\text{-}OSR$, o algoritmo provê p_i com todos os nós alcançáveis do seu componente fortemente conectado mais os nós alcançáveis de outras componentes (o que inclui pelo menos todos os nós na componente poço).

Neste ponto, $R[i].end\_col$ é ajustado para 1, anunciando a outros processos que p_i já terminou a sua prospecção e que o estado corrente de $Known[i]$ é seu estado final. Esta informação sobre cada processo será necessária para executar o próximo algoritmo – SINK().

Descrição. Na fase INIT, p_i invoca i.PD para obter o conhecimento inicial sobre o grupo, armazenando-o em $Known.[i]$ (linha 9). Para notificar outros processos que tal informação já está disponível, p_i ajusta $R[i].end\_pd$ para 1 (linha 10). Após a execução da linha 11, p_i é o único processo na variável privada $included_i$.

Na fase IMPROVEMENT, p_i executa dois laços. No laço mais interno (**for** – linha 13), p_i insere no seu conjunto de conhecidos a visão de cada p_j que ele conhece e que já recebeu a informação de j.PD (linhas 13-16). Cada vez que ele faz isso, inclui p_j em seu conjunto local $incuded_i$ (linha 17). Esse procedimento é repetido para todo p_j que for achado em $Known[i]$ cuja visão ainda não tiver sido inserida em $Known[i]$ (linha 13) e enquanto a condição do laço externo não é satisfeita.

O laço externo (**while** – linha 12), controla a terminação do algoritmo, garantindo que p_i permanece juntando ao seu conhecimento a visão de todos os seus conhecidos, a menos, possivelmente, do número de processos faltosos.

Algorithm 1 COLLECT() process p_i

constant:
1: f: int // *upper bound on the number of crashes*

shared variables:
2: $Known[i] \leftarrow \varnothing$: set of nodes
3: $R[i]$: multivalued $1WMR$ register
4: $R[i].end\_pd \leftarrow 0$: binary value
5: $R[i].end\_col \leftarrow 0$: binary value
6: $R[i].decision \leftarrow \perp$: value

local variables:
7: $included_i, known_i \leftarrow \varnothing$: set of nodes
8: $pd_i^j \leftarrow 0$: binary value

procedure:

INIT:
9: $Insert(Known[i], i.\mathsf{PD})$;
10: $write(R[i].end\_pd, 1)$;
11: $included_i \leftarrow \{i\}$;

IMPROVEMENT:
12: **while** $(|included_i| < |(Known[i])| - f)$ **do**
13: **for each** $j \in (Known[i] \setminus included_i)$ **do**
14: $read(R[j].end\_pd, pd_i^j)$
15: **if** $(pd_i^j = 1)$ **then**
16: $Insert\ (Known[i], Known[j])$;
17: $included_i \leftarrow included_i \cup \{j\}$;
18: $write(R[i].end\_col, 1)$;
19: **return** ;

3.6.2 Corretude do algoritmo

As seguintes propriedades são asseguradas com a execução do Algoritmo 1 (COLLECT()), em um sistema assíncrono estendido com um detector de participantes k-OSR, onde até $f < k$ processos podem falhar por parada.

Lema 3.1. *Seja p_i um processo correto. Se o processo p_j é alcançável a partir de p_i, em G_{di}, por k caminhos disjuntos nos nós, então, em algum instante ocorrerá que $p_j \in Known[i]$, a despeito de $f < k$ falhas.*

Prova:

(A) No início de COLLECT(), $Known[i] = i.\mathsf{PD}$ (linha 9), então todo p_j alcançável a partir de p_i, com uma distância em G_{di} igual a 1, está em $Known[i]$.

(B) Na execução do *laço* **while** (linha 12), p_i adiciona a $Known[i]$ a visão de todo $p_j \in Known[i]$ (linha 16) (que já incluiu j.PD em $Known[j]$ – linha 15). Nesse caso, p_i adiciona p_j a $included_i$ (linha 17).
O **while** só termina quando $included_i$ contiver todos os processos corretos conhecidos por p_i, a menos, possivelmente, de f processos (que podem ter falhado) (linha 12). Portanto, ao final do algoritmo (linha 19), $\forall p_j \in Known[i]$, desde que p_j não tenha falhado antes de inserir a informação recebida de seu detector de participantes, pelo menos $j.PD$ terá sido incluído em $Known[i]$.

(C) Por suposição, existem pelo menos k caminhos disjuntos nos nós pelos quais p_i alcança $p_j \in G_{di}$ (Definição 3.4). Então, pela Observação 3.4.1, a falha de $f < k$ nós deixa pelo menos um caminho $P : p_i = p_0, p_1, ..., p_{y-1}, p_y = p_j$ de nós corretos de p_i a p_j. Será provado por indução sobre k que em algum instante $p_k \in Known[i], \forall k = 1, 2, ..., y$.

(1) A base $k = 1$ é trivial, uma vez que $p_1 \in i$.PD.

(2) Assuma que a alegação é válida para todo $p_k, k < y$.

Uma vez que pela hipótese de indução, $p_{y-1} \in Known[i]$, então, de (B), $Known[y-1]$ acabará sendo adicionado a $Known[i]$. Como $p_y \in (y-1).PD$, tem-se que $p_y \in Known[y-1]$. Então, $p_y = p_j$, em algum instante, acabará sendo incluído em $Known[i]$. $_{Lema 3.1}\square$

Lema 3.2. *O Algoritmo 1* (COLLECT()) *satisfaz as seguintes propriedades, a despeito de $f < k$ falhas:*

(1) Segurança (*Safety*). Para todo nó correto p_i, $Known[i]$ é tal que:

(i) $p_i \in G_{sink} \Leftrightarrow Known[i] = G_{sink}$;

(ii) $p_i \notin G_{sink} \Leftrightarrow Known[i] \supsetneq G_{sink}$.

(2) Terminação. Todo nó correto p_i termina a execução.

Prova:

(1) Prova de Segurança

Caso 1: $p_i \in G_{sink}$.

$[\Rightarrow]$ $p_i \in G_{sink} \rightarrow Known[i] = G_{sink}$.

(A) Por definição, G_{sink} é k-fortemente conectado, então $\forall p_i, p_j \in G_{sink}$, a Observação 3.4.1 se mantém. Assim, pelo Lema 3.1, em algum instante $p_j \in Known[i]$.

(B) Considere a construção de $Known[i], \forall p_i \in G_{sink}$.

(B.1) Por definição, não existe caminho em G_{di} de p_i a p_j, $\forall p_j \notin G_{sink}$. Então, $p_j \notin i$.PD.

(B.2) A cada iteração do laço **while** (linha 12), p_i agrega a $Known[i]$ o conhecimento de cada p_k correntemente em $Known[i]$.
Portanto, nenhum p_j fora de G_{sink} será inserido em qualquer $Known[i]$.

De (A) e (B), tem-se que $Known[i] = G_{sink}$. ⊙

[⇐] $Known[i] = G_{sink} \rightarrow p_i \in G_{sink}$.

Por hipótese, $Known[i] = G_{sink}$. Suponha, por contradição, que $p_i \notin G_{sink}$. Uma vez que $p_i \in Known[i]$, $Known[i] \neq G_{sink}$, contradizendo a hipótese. ⊙

Caso 2: $p_i \notin G_{sink}$.

[⇒] $p_i \notin G_{sink} \rightarrow Known[i] \supsetneq G_{sink}$.

(A) Pela Definição 3.4, existem pelo menos k caminhos disjuntos nos nós de qualquer $p_i \notin G_{sink}$ para qualquer $p_j \in G_{sink}$. Portanto, $\forall p_j \in G_{sink}$, $p_j \in Known[i]$ (Lema 3.1) e então $Known[i] \supset G_{sink}$.

(B) Uma vez que $p_i \notin G_{sink}$ e $p_i \in Known[i]$, $Known[i] \neq G_{sink}$.

De (A) e (B), $Known[i] \supsetneq G_{sink}$ ⊙
[⇐] $Known[i] \supsetneq G_{sink} \rightarrow p_i \notin G_{sink}$.

Suponha, por contradição, que $p_i \in G_{sink}$. Pela definição da classe k-OSR (Def. 3.4), não existe caminho de p_i para $\forall p_j \notin G_{sink}$. Além disso, do Caso 1, tem-se que $Known[i] = G_{sink}$, contradizendo a hipótese. ⊙

A partir dos Casos 1 e 2, faz-se as seguintes observações:

Observação 3.6.1. *Se $p_i \in G_{sink}$, então $\forall p_j \in Known[i], p_i \in Known[j]$.*

Observação 3.6.2. *Se $p_i \notin G_{sink}$, então $\forall p_j \in G_{sink}$: $p_j \in Known[i]$, e $p_i \notin Known[j]$.*

(2) Prova de Terminação

(A) A condição de parada do algoritmo é $|included_i| \geq (|Known[i]| - f)$ (linha 12). $Known[i]$ é limitado pelo número de processos em uma execução n. O conjunto $included_i$ representa os nós em $Known[i]$ cujos conhecidos foram adicionados à visão de p_i (linhas 13-17); portanto $included_i \subseteq Known[i]$.

(B) Inicialmente, $included_i$ tem um elemento (linha 11).

(C) Ao executar o laço **while** (linhas 12-17), todo $p_j \in Known[i]$ é cotejado e todo elemento em sua visão $Known[j]$ que ainda não está em $Known[i]$ é incluído em $Known[i]$ enquanto que o próprio p_j é incluído em $included_i$. Como G_{di} é conectado por definição, tem-se $|Known[j]| > 1$, $\forall j$. Portanto, é possível que $Known[i]$ cresça mais que $included_i$ nas primeiras iterações do laço **while**, mantendo a condição exigida pelo laço: $|included_i| < (|Known[i]| - f)$.

(D) Entretanto, uma vez que $|Known[i]|$ é limitado, ele para de crescer, isto é, ocorrem iterações em que os nós pertencentes à visão do processo p_j sendo considerado já foram incluídos em $Known[i]$. Por exemplo, considere o caso em que $Known[i]$ atinge a maior cardinalidade possível, isto é, $|Known[i]| = n$. A partir desse instante, todo $p_j \in Known[i]$ que ainda não foi considerado (linhas 13-17)) será incluído em $included_i$ sem que $|Known[i]|$ sofra qualquer alteração. Dessa maneira, $included_i$ permanece crescendo até que a condição da linha 12 não é mais satisfeita, fazendo com que p_i saia do laço, terminando sua execução. $_{Lema 3.2}\square$

3.7 O ALGORITMO SINK: DETERMINANDO OS NÓS NA COMPONENTE POÇO

O Algoritmo 2 – SINK() – usa o conhecimento adquirido pelos processos com o algoritmo COLLECT() para determinar quais processos pertencem à componente poço. Essa definição é baseada no fato de que os nós no poço se conhecem entre si, enquanto que os nós fora do poço conhecem todos os nós do poço, mas os nós do poço não os conhecem. Isto é o que atestam as observações 3.6.1 e 3.6.2. A informação da localização de p_i em G_{di} será usada no algoritmo CONSENSUS().

Descrição. Na fase INIT, p_i inicializa o conjunto de processos checados com sua própria identidade e chama COLLECT() para construir sua visão do sistema, a qual é armazenada em $Known[i]$ (linha 10).

Na fase VERIFICATION, p_i verifica se existe em seu conjunto de conhecidos algum p_j que não o conhece. Se isso ocorre, o algoritmo retorna 0, indicando que p_i não está no poço. Caso contrário, retorna 1. Enquanto o número de processos verificados não atinge o número de conhecidos de p_i, a menos de f que podem ser faltosos (linha 11), p_i avalia o conjunto $Known[j]$ de todo $p_j \in Known[i]$ que já tiver terminado COLLECT() (i.e., $R[j].end\_col = 1$) (linhas 12-14). Se p_j ainda não tiver terminado COLLECT(), a verificação continua com o próximo p_j ainda não checado. O algoritmo termina de uma das seguintes maneiras:

1. p_i não está na componente poço, o que é detectado se $\exists p_j : p_i \notin Known[j]$, nesse caso, p_i retorna 0 (linhas 16-17); ou

2. p_i pertence ao poço, isto é, p_i checou $Known[j]$ para todo $p_j \in Known[i]$ e o predicado da linha 16 sempre foi avaliado falso até que a condição do **while** deixou de ser satisfeita. Então ao sair do laço, p_i retorna 1 (linha 19).

3.7.1 Corretude do algoritmo

Lema 3.3. *Em um sistema assíncrono estendido com um PD k-OSR, o Algoritmo 2* (SINK()), *com a ajuda do algoritmo (COLLECT()), satisfaz as seguintes propriedades, a despeito de $f < k$ falhas:*

(1) Segurança (*Safety*). Para todo nó correto p_i:

(i) $p_i \in G_{sink} \Leftrightarrow p_i$ retorna 1;

Algorithm 2 SINK() process p_i

constant:
1: f: int // *upper bound on the number of crashes*

shared variables:
2: $Known[i] \leftarrow \varnothing$: set of nodes
3: $R[i]$: multivalued $1WMR$ register
4: $R[i].end\_pd \leftarrow 0$: binary value
5: $R[i].end\_col \leftarrow 0$: binary value
6: $R[i].decision \leftarrow \perp$: value

local variables:
7: $checked_i, known_j \leftarrow \varnothing$: set of nodes
8: $endcol_i \leftarrow 0$: binary value

procedure:

INIT:
9: $checked_i \leftarrow \{i\}$;
10: COLLECT();

VERIFICATION:
11: **while** $(|checked_i| < |(Known[i])| - f)$ **do**
12: **for each** $j \in ((Known[i]) \setminus checked_i)$ **do**
13: $read(R[j].end\_col, endcol_i)$;
14: **if** $(endcol_i = 1)$ **then**
15: $get(Known[j], known_j)$;
16: **if** $(i \notin known_j)$ **then**
17: **return** 0;
18: $checked_i \leftarrow checked_i \cup \{j\}$;
19: **return** 1;

(ii) $p_i \notin G_{sink} \Leftrightarrow p_i$ retorna 0.

(2) Terminação. Todo nó correto p_i termina a execução, decidindo se pertence ou não a G_{sink}.

Prova:

(1) Prova de Segurança

CASO 1: $p_i \in G_{sink}$.

[$\Rightarrow$] $p_i \in G_{sink} \rightarrow p_i$ retorna 1.

Da Observação 3.6.1, $\forall p_i \in G_{sink}$, o predicado da linha 16 será avaliado falso para todo $p_j \in Known[i]$. Então, após sair do laço **while**, p_i retorna 1 (linha 19).

[$\Leftarrow$] p_i retorna 1 $\rightarrow p_i \in G_{sink}$.

Suponha, por contradição, que $p_i \notin G_{sink}$. Então, $\exists p_j \in G_{sink}$, p_j correto, tal que:

(A) Pela Definição 3.4 e Lema 3.1, $p_j \in Known[i]$;

(B) $p_j \in Known[i]$ e $p_i \notin Known[j]$ (Observação 3.6.2).

Por (A) e (B), o predicado da linha 16 é satisfeito, e, em seguida, SINK() retorna 0, contradizendo a suposição. $\odot$.

Caso 2: $p_i \notin G_{sink}$.

[$\Rightarrow$] $p_i \notin G_{sink} \rightarrow p_i$ retorna 0.

Da Observação 3.6.2, $\exists p_j \in G_{sink}$ tal que $p_j \in Known[i]$ e $p_i \notin Known[j]$. Então, o predicado da linha 16 é avaliado verdadeiro e p_i retorna 0.

[$\Leftarrow$] p_i retorna 0 $\rightarrow p_i \notin G_{sink}$.

Suponha, por contradição, que $p_i \in G_{Sink}$. A prova segue diretamente do caso 1, [$\Rightarrow$]. Então, p_i retorna 1, contradizendo a hipótese.

(2) Prova de Terminação.

p_i irá examinar todo p_j em sua visão local (exceto por um número f de processos que podem ser faltosos), desde que p_j já tenha terminado COLLECT() (linhas 11–14). À medida que verifica um p_j, p_i o inclui no conjunto $checked_i$, (linha 18), se não retornar antes (linha 17). Note-se que $Known[i]$ é limitado por n e que $checked_i \subseteq Known[i]$. Tem-se então:

(A) se $p_i \notin G_{sink}$, dado que existe pelo menos um processo correto p_j em G_{sink}, p_i observa que ele não pertence a $Known[j]$, concluindo que ele próprio não pertence ao poço. Então p_i termina o algoritmo retornando 0 na linha 17. (Note-se que $p_j \in Known(i)$ – Observação 3.6.2).

(B) se, por outro lado, $p_i \in G_{sink}$, tem-se que $Known[i] = G_{sink}$ (Lema 3.2). Então para todo p_j que p_i examina, verifica que ele está na visão de p_j. Então ele inclui p_j em $checked_i$ e passa a examinar outro p_j. Uma vez que $Known[i]$ é limitado, chegará um instante em que $|checked_i| = |Known[i]| - f$ e p_i, saindo do laço **while**, retorna 1 na linha 19. $_{Lema 3.3}\square$

3.8 O ALGORITMO CONSENSUS: DECIDINDO UM VALOR

O Algoritmo 3 – CONSENSUS() – é executado por cada processo para decidir por um valor entre os valores propostos. Uma vez que os nós na componente poço G_{sink} são os únicos acessíveis por todos os outros nós no grafo G_{di}, apenas esses nós participam da

decisão. Como esses nós compartilham exatamente a mesma visão do sistema, o problema é reduzido ao modelo clássico em que todo o grupo é conhecido. Assim, pode-se resolver o consenso com a ajuda do detector Ω, desde que haja pelo menos um nó correto em G_{sink} (GUERRAOUI; RAYNAL, 2007). Após a decisão ser tomada pelos nós no poço e armazenada nos registradores apropriados, os nós que estão nas outras componentes podem recuperar o valor decidido.

Descrição. Na fase inicial, todos os nós invocam SINK() para construir sua visão do sistema e determinar se eles pertencem à componente poço ou não (linha 11). Os nós do poço invocam *Agreement*() (linha 12), a fim de decidir por um valor v, enquanto os demais nós, nas outras componentes, invocam *GetDecision*() (linha 14) para obter a decisão. Inicialmente, a decisão de cada processo p_i é nula (linha 10) ($\perp$ denota um valor padrão que não pode ser decidido por um processo).

Para decidir, os nós no poço executam um protocolo de consenso subjacente (linhas 15-16), que pode ser qualquer um dos propostos até então, por exemplo, aqueles que utilizam um detector Ω para atingir o consenso (GUERRAOUI; RAYNAL, 2007). Assim que cada $p_i \in G_{sink}$ decide, publica sua decisão v em $R[i].decision$ (linha 17) e retorna v para a aplicação.

Os nós fora de G_{sink} executam *GetDecision*(). Um processo p_i mantém-se lendo os registradores dos processos conhecidos, a fim de aprender sobre a decisão tomada (linhas 20-21). Uma vez que obtém uma decisão, ele a escreve em $R[i].decision$ (linhas 22-23), ajudando os outros processos a tomarem a mesma decisão. Em seguida, ele retorna v terminando sua participação.

3.8.1 Corretude do Algoritmo

Lema 3.4. *O Algoritmo 3* (CONSENSUS()) *resolve SM FT-CUP uniforme, a despeito de $f < k$ falhas, assumindo pelo menos um nó correto na componente poço do grafo G_{di}, induzido por um* PD *k-OSR.*

Prova:

(1) Prova de Validade.

É baseada na propriedade validade do algoritmo de consenso subjacente e no fato de que a decisão é tomada a partir dos valores propostos pelos participantes do poço (linha 15). ⊙

(2) Prova de Terminação.

Para provar que cada nó correto p_i decide, deve-se mostrar que p_i termina executando a linha 18 ou 24 de CONSENSUS(). A partir do **if** da linha 11, p_i pode tomar uma de duas direções:

Caso 1: $p_i \in G_{sink}$

p_i executa as linhas 11 e 12 e, em seguida, chama um algoritmo subjacente para realizar o consenso. Uma vez que há pelo menos um nó correto no poço e, devido à propriedade Terminação deste algoritmo subjacente, p_i consegue tomar uma decisão (linhas 15-16).

Algorithm 3 CONSENSUS(*value*)

constant
1: f: int // *upper bound on the number of crashes*

input
2: *value* // *proposition value to the consensus*

shared variables
3: $Known[i] \leftarrow \varnothing$: set of nodes
4: $R[i]$: multivalued $1WMR$ register
5: $R[i].end\_pd \leftarrow 0$: binary value
6: $R[i].end\_col \leftarrow 0$: binary value
7: $R[i].decision \leftarrow \perp$: value
8: $R[i].end\_pd \leftarrow 0$: binary value

local variables
9: $d \leftarrow \perp$: value // *decision value for the consensus*

procedure

***** All Nodes *****
10: $d \leftarrow \perp$;
11: **if** (SINK()) **then**
12: *Agreement*();
13: **else**
14: *GetDecision*();

***** Node in Sink *****
Agreement() // *Underlying Consensus*
15: Consensus.propose(*value*);
16: **upon** Consensus.decide(v);
17: $write(R[i].decision, v)$;
18: **return** v;

***** Node Not in Sink *****
GetDecision() // *Get Decision*
19: **while** $(d = \perp)$ **do**
20: **for each** $(j \in Known[i])$ **do**
21: $read(R[j].decision, d)$;
22: **if** $(d \neq \perp)$ **then**
23: $write(R[i].decision, d)$;
24: **return** v;

Então ele a escreve em $R[i].decision$, retornando em seguida para a aplicação (linhas 17-18). $\odot$

Caso 2: $p_i \notin G_{sink}$

(A) p_i executa as linhas 11 e 14, para obter uma decisão a partir de algum registrador. Assim, p_i começa a ler $R[j].decision$ para todo $p_j \in Known[i]$ (linhas 20-21). Quando ele encontra um valor válido ($\neq \perp$), decide por esse valor e retorna para a aplicação (linhas 22-24).

(B) Do Caso 1, pelo menos um nó do poço escreve uma decisão v em seu registrador.

(C) Da Observação 3.6.2, $\forall p_j \in G_{sink}$: $p_j \in Known[i]$.

De (A), (B), e (C), tem-se que p_i decide baseado na decisão obtida em algum registrador. $\odot$

(3) Prova de Acordo Uniforme.

É assegurada pela propriedade *acordo uniforme* do algoritmo subjacente usado. Assim, todo nó $p_j \in G_{sink}$ concorda com o mesmo valor de decisão v (linha 16) e escreve esse valor em seu registrador compartilhado $R[j].decision$ (linha 17). Portanto, todo nó $p_i \notin G_{sink}$ vai ler o mesmo valor de $v \neq \perp$ em algum registrador e decidir esse valor (linhas 20-23). $\square_{Lema 3.4}$

3.9 COMPLEXIDADE DOS ALGORITMOS

Todo o conjunto de protocolos apresentados neste trabalho para resolver SM FT-CUP usa n *registradores regulares* e n *conjuntos*, um para cada participante do sistema, mas n não precisa ser conhecido a priori. A identidade de um processo, portanto, é limitada por $log_2(n)$ bits. Um conjunto associado a um processo armazena a identidade de todos os processos conhecidos dele. Seu tamanho, portanto, é limitado por $nlog_2(n)$.

Cada registrador possui três campos: *end_pd*: binário, *end_col*: binário, e *decision*: domínio de valores propostos pelos processos. Logo, o tamanho do registrador depende do domínio de valores de decisão.

No algoritmo COLLECT(), cada p_i executa uma única operação de leitura sobre o registrador de cada processo p_j ($R[j].end\_pd$) em $Known[i]$ (linha 14). Como o número de nós neste conjunto é limitado por n, o número de leituras é limitado pelo mesmo valor. Também são executadas duas operações de escrita, uma sobre $R[i].end\_pd$ (linha 10) e uma sobre $R[i].end\_col$ (linha 18). Além das operações *read* e *write* explícitas no algoritmo, são executadas operações de inserção em $Known[n]$ (linha 16). O número dessas operações também é limitado pela cardinalidade de $Known[i]$, isto é, n.

No algoritmo SINK(), nenhuma operação de escrita é realizada. Inicialmente, todo $R[j].end\_col = 0, \forall j$ e cada p_i executa operações de leitura sobre $R[j].end\_col, \forall j \in Known[i]$ até que seu valor tenha sido ajustado para 1 (linhas 12-14). Então, p_i obtém os valores em $Known[j]$ (linha 15), isto é, o número de operações *get* executas por p_i é limitado por n.

Em CONSENSUS(), cada p_i só escreve no registrador uma vez, quando decide (linha 17 ou 23). Se p_i é um nó do poço, o número de operações de leitura e escrita depende do algoritmo de consenso utilizado. Por outro lado, se p_i não está no poço, permanece lendo o arranjo de registradores (cada $R[j].decision$) até que uma decisão seja obtida (linha 21). Desse modo, o número de leituras é indeterminado.

3.10 CONSIDERAÇÕES

O desenho dos algoritmos revela que o detector de participantes k-OSR e o detector de líder Ω reúnem as condições suficientes (conectividade do conhecimento e sincronia, respectivamente) para atingir o consenso em sistemas assíncronos de memória compartilhada, considerando que $f < k$, e que pelo menos um nó no poço é correto.

É importante observar que o algoritmo COLLECT() não precisa de objetos atômicos/linearizáveis. Ele usa um laço **while** para obter e inserir identidades de processos no conjunto $Known$, mas se um **get()** concorrente a um **insert()** não obtém os novos elementos inseridos por esta operação, em algum momento, se o processo não falha, estes elementos serão obtidos em operações **get()** subsequentes. As condições de terminação e resiliência estabelecidas no algoritmo garantem que as novas informações obtidas necessárias a partir de um **get()** acabarão sendo adquiridas.

Os outros algoritmos (SINK() e CONSENSUS()) só obtêm elementos do conjunto $Known$ (eles nunca inserem), e portanto eles não requerem objetos linearizáveis. O mesmo se aplica aos campos binários *end_pd* e *end_col* do registrador. Eles são escritos apenas uma vez em COLLECT(), indicando o término da consulta ao PD com a escrita do valor 1; e o término do algoritmo, respectivamente. Em seguida, *end_pd* é lido em COLLECT() mesmo e *end_col* em SINK(). Se alguma dessas leituras sobrepõe a escrita em COLLECT(), o pior que pode acontecer é a execução de novas iterações até que uma leitura seja iniciada após o fim da escrita.

O caso do campo *decision* é semelhante. Ele é utilizado nas funções $Agreement()$ e $GetDecision()$ de CONSENSUS() e, tão logo a escrita do valor v de uma decisão termine, em $Agreement()$, este valor pode ser lido em $GetDecision()$. Em conclusão, n objetos não linearizáveis como o conjunto e n registradores regulares, são suficientes para construir os algoritmos, garantindo sua segurança (*safety*). Na execução do consenso subjacente em $Agreement()$, o tipo e o número de registradores utilizados dependem do algoritmo adotado.

3.11 CONCLUSÃO

Foi apresentado um conjunto de protocolos para resolver o consenso tolerante a falhas em um sistema assíncrono com participantes desconhecidos, considerando o modelo de memória compartilhada para comunicação entre processos – SM FT-CUP. A abordagem utilizada é baseada no uso de detectores de participantes que proveem os processos com um conhecimento inicial sobre os participantes da computação. Tal conhecimento pode ser modelado como um grafo da conectividade do conhecimento cujas propriedades definem a classe do detector de participantes. Os algoritmos funcionam corretamente para

sistemas equipados com um detector de participantes da classe k-OSR e um detector de falhas Ω, considerando $f < k$ falhas.

Os resultados alcançados são um passo inicial para a criação de soluções práticas em serviços de armazenamento de dados com topologia arbitrária, como as *redes de área de armazenamento*, que interligam vários discos conectados a uma rede, os quais podem ser acessados por quaisquer processos ou aplicações de nuvem comunicando-se sobre armazenamento compartilhado. Eles são igualmente úteis em sistemas multinúcleo, inclusive focando em sua evolução para sistemas *exascale*.

O conjunto de protocolos apresentado depende de um algoritmo de consenso para realizar o consenso entre os nós do poço. A partir destes, uma proposta interessante seria desenvolver um algoritmo de consenso que pudesse então ser invocado pelos nós do poço, estendendo o conjunto de protocolos oferecido. Este algoritmo foi de fato desenvolvido e é apresentado no próximo capítulo. Com o objetivo de proporcionar maior flexibilidade, o algoritmo é genérico quanto ao detector de falhas, podendo ser instanciado com um detector Ω ou $\diamondsuit\mathcal{S}$.

Capítulo

4

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 capítulo, é apresentado um algoritmo de 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.

CONSENSO GENÉRICO PARA SISTEMA DINÂMICO DE MEMÓRIA COMPARTILHADA

4.1 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 (CHANDRA; TOUEG, 1996) 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. 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 se comunicam através de uma memória compartilhada, apesar de sua importância.

Como já foi dito nos capítulos anteriores, não existe solução determinística para o consenso em sistemas assíncronos sujeitos a falhas (FISCHER; LYNCH; PATERSON, 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; TOUEG, 1996). O oráculo líder, Ω, fornece aos

processos a identidade de um processo (supostamente) correto (CHANDRA; HADZILACOS; TOUEG, 1996). Ao cabo de um tempo, a identidade de um mesmo processo correto é retornada para todos os processos. As classes de detectores $\diamondsuit\mathcal{S}$ e Ω são as mais fracas que permitem realizar o consenso em sistemas de passagem de mensagens (CHANDRA; HADZILACOS; TOUEG, 1996) ou de memória compartilhada (LOUI; ABU-AMARA, 1987).

Apesar de viabilizarem o consenso, esses detectores não são confiáveis e podem cometer erros. A despeito disso, os protocolos de consenso baseados nesses detectores devem ser *indulgentes* (GUERRAOUI, 2000) para com eles. Isso quer dizer que os algoritmos devem conviver pacificamente com os erros do detector, garantindo que suas propriedades de segurança (ou *safety*) sejam mantidas durante períodos de instabilidade e assincronia do ambiente. A terminação (ou *liveness*) do algoritmo será então viabilizada pelo detector quando tal período instável tiver terminado.

No capítulo 3 foi apresentado um conjunto de algoritmos para resolver o SM FT-CUP – consenso com participantes desconhecidos tolerante a falhas em memória compartilhada. A solução considera o uso de um detector de participantes da classe k-OSR que provê os processos com um conhecimento inicial sobre o sistema. Com a execução dos algoritmos, o problema é, ao final, reduzido ao problema clássico do consenso entre os nós do componente poço do grafo de conhecimento do sistema, isto é, o conjunto de participantes é conhecido.

Neste capítulo, é apresentado 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 $\diamondsuit\mathcal{S}$ ou Ω. Alguns trabalhos fornecem soluções para sistemas estendidos ou com $\diamondsuit\mathcal{S}$ (LO; HADZILACOS, 1994) ou com Ω (DELPORTE-GALLET; FAUCONNIER, 2009), mas não foi encontrado na literatura um algoritmo com ambos.

O desenvolvimento do algoritmo passou por duas fases. Na primeira, foram desenvolvidas duas versões, apresentadas em (KHOURI; GREVE, 2013a) e (KHOURI; GREVE, 2013b), que estendem o conjunto de algoritmos do capítulo 3 e objetivam resolver o consenso entre os nós do componente poço. Portanto, por si só, o algoritmo não suporta dinamismo, já que considera o conhecimento da cardinalidade n do conjunto de processos do sistema. Até onde se sabe, este é o primeiro algoritmo de consenso genérico para memória compartilhada.

Na segunda fase, o algoritmo obtido não assume o conhecimento de n, e então pode ser usado de maneira plena e direta em sistemas dinâmicos. O resultado é então o primeiro protocolo de consenso genérico quanto ao oráculo para sistemas de memória compartilhada, que não exige o conhecimento prévio de n. Considerando a sincronia mínima necessária para resolver o consenso, o algoritmo é ótimo, já que é baseado nos detectores $\diamondsuit\mathcal{S}$ e Ω. Também é ótimo no que diz respeito à resiliência, porque tolera qualquer número de falhas de processos $(n - 1)$, desde que pelo menos um processo permaneça ativo para sempre.

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 do consenso pode ser melhor adaptada às car-

acterísticas particulares de cada ambiente; principalmente quando a implementação do detector serve a muitas aplicações. Desse modo, as aplicações existentes que já estão executando sobre os detectores $\diamondsuit\mathcal{S}$ ou Ω podem ser portadas mais facilmente.

O restante do capítulo está organizado como segue: a Seção 4.2 descreve o modelo do sistema. Na Seção 4.3 são relembradas as definições referentes ao consenso e oráculos. As seções 4.4, 4.5 e 4.6 apresentam o algoritmo genérico, as provas de corretude e uma análise da complexidade, respectivamente. A Seção 4.7 apresenta alguns trabalhos relacionados e a Seção 4.8 conclui o capítulo.

4.2 MODELO DO SISTEMA

O sistema dinâmico é composto por um conjunto infinito de processos que se comunicam através de uma memória compartilhada. Qualquer número de processos menos 1 pode falhar (*crashing*), parando prematura ou deliberadamente (por exemplo, saindo arbitrariamente do sistema). Não são feitas 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, assume-se a existência de um relógio global que não é acessível aos processos, cuja faixa de tiques, $\mathcal{T}$, é o conjunto dos números naturais.

4.2.1 Processos

Considera-se o *modelo de chegada infinita com concorrência limitada* (*bounded concurrency*) (GAFNI; MERRITT; TAUBENFELD, 2001), isto é, o conjunto de participantes do sistema, Π_{BC}, tem um número infinito de processos que podem entrar e sair em instantes arbitrários, mas em cada execução existe um número limitado de processos, n. Portanto, em cada execução, a cardinalidade de Π_{BC} é limitada, embora seja desconhecida. 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 texto, um processo pode ser denotado por seu nome ou por sua identidade.

Tal como em (BALDONI; BONOMI; RAYNAL, 2012), assume-se que o sistema subjacente se encarrega de gerenciar a entrada e saída de processos na computação, bem como sua conectividade (ver Seção 2.7.1). Quando um processo p_i deseja entrar na computação, ele requisita uma operação $join()$ que lhe fornece um estado consistente dos registradores no sistema ao passo que provê, aos demais processos, o acesso de leitura aos registradores de p_i. Após o término de $join()$, p_i é considerado *ativo* até que falhe ou deixe o sistema. O conjunto de processos ativos é assim definido:

Definição 4.1 (ACTIVE)**.** $ACTIVE(t) = \{p \in \Pi_{\mathsf{BC}} : p \text{ está ativo no instante } t\}$.

A despeito da movimentação de entrada e saída de processos, a existência de um conjunto de processos estáveis num sistema dinâmico é necessária para que se possa realizar alguma computação útil (MOSTEFAOUI et al., 2005). A noção de *estável* refere-se a um processo que entra no sistema e permanece ativo durante toda computação. O parâmetro α representa o limite inferior para a cardinalidade do conjunto de processos estáveis.

Definição 4.2 (STABLE). $STABLE = \{p_s \in \Pi_{\mathsf{BC}} : \exists t \in \mathcal{T}, \forall t' \geq t : p_s \in ATIVO(t')\}$

No modelo em questão, admite-se a existência de pelo menos um processo estável no sistema, isto é, $\alpha \geq 1$. Na realidade, para a convergência do algoritmo é suficiente que um processo permaneça ativo até que escreva uma decisão v em seu registrador e retorne v para a aplicação.

Como foi dito na Introdução, as primeiras versões do algoritmo foram destinadas a resolver o consenso entre os nós do componente poço. Nesse caso, a questão do conhecimento dos participantes do sistema é tratada pelos algoritmos COLLECT e SINK que juntos identificam os nós do poço, os quais são então encarregados de realizar o consenso entre si, disseminando, finalmente, sua decisão para os demais processos. Assim, para estas versões, é considerado o modelo estático, em que o conjunto de participantes é conhecido e fixo.

4.2.2 Memória Compartilhada

A memória compartilhada consiste de um arranjo $R[n]$ de registradores regulares multivalorados do tipo 1-*escritor-múltiplos-leitores* (ou $1WMR$) que se comportam corretamente, conforme definido na Seção 2.2.1.1. Cada registrador possui três campos que serão descritos posteriormente.

Um registrador $R[i]$ está associado ao processo p_i, que é seu único escritor, e pode ser lido por qualquer p_j que entra na computação.

4.3 CONSENSO BASEADO EM ORÁCULO

Para contornar a impossibilidade do consenso (ver Seção 2.5), a estratégia seguida aqui é estender o sistema com um oráculo que forneça dicas sobre as falhas de processos. Uma vez que os oráculos $\diamondsuit\mathcal{S}$ e Ω são minimais, no sentido de que fornecem o menor teor de sincronismo que permite a realização do consenso, qualquer um deles poderia ser utilizado na solução. Com o intuito de oferecer uma solução genérica, o algoritmo produzido pode ser instanciado com qualquer um dos dois oráculos, $\diamondsuit\mathcal{S}$ ou Ω.

Para relembrar a definição do consenso uniforme e dos oráculos, as propriedades sobre as quais cada um deles é definido são transcritas a seguir (seções 2.5 e 2.6).

Consenso Uniforme. Cada processo p_i propõe um valor v_i e todo processo que decide, decide um mesmo valor v, entre os propostos, de modo que as seguintes propriedades são satisfeitas:

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.

Detector de Falhas $\diamondsuit\mathcal{S}$. Quando invocado por um processo, um detector $\diamondsuit\mathcal{S}$ fornece uma lista de processos suspeitos, satisfazendo as seguintes propriedades:

Completude forte (*Strong completeness*): Após um tempo, todo processo faltoso ou que abandona o sistema 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. Um oráculo da classe Ω, quando invocado por um processo, retorna a identidade de um processo considerado correto. A classe Ω satisfaz à seguinte propriedade:

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

Vale ressaltar que os instantes de estabilidade garantidos pelas propriedades de Ω e $\diamondsuit\mathcal{S}$ não são conhecidos pelos processos. Entretanto, a existência desses instantes permite garantir a terminação dos algoritmos de consenso baseados neles.

4.4 ALGORITMO GENÉRICO – GENERICON

O algoritmo GENERICON $(v_i, \mathcal{D})$ é executado por todo processo p_i para decidir um valor. Ele consiste de um laço infinito que recebe como argumento o valor inicial de p_i e pode ser instanciado 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, r_i, correntemente executada por p_i, é atualizado seguindo uma sequência estritamente crescente.

Em cada rodada, todo p_i define um processo como proponente ($proposer_i$), de acordo com o oráculo instanciado, que tentará impor uma decisão, a qual será adotada pelos demais. Se o oráculo é $\diamondsuit\mathcal{S}$, a escolha segue o paradigma do coordenador rotativo, de modo que cada processo, um por vez, tem a chance de ser o proponente. Se o oráculo é Ω, o proponente é o próprio líder retornado pelo oráculo. Quando o proponente é considerado suspeito, p_i desiste de esperar por sua decisão e define um novo 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 processos corretos. Em ambos os casos, o resultado é a coexistência de vários proponentes tentando impor sua decisão. 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; RAYNAL, 2004). Assim, quando um proponente p_i, numa rodada r_i, percebe a presença de outro proponente p_j (ou mais de um) na mesma rodada que ele ou numa rodada mais avançada, $r_j \geq r_i$, desiste de tentar decidir.

Enquanto novos proponentes são continuamente definidos, aqueles anteriores (com números de rodada menores), continuamente desistem de tentar impor sua decisão, e

nesse período de instabilidade, 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.

4.4.1 Registradores Compartilhados

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 $R[i].value$: est, pro ou dec. Inicializado com $\perp$.

As operações executadas sobre os registradores são:

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

Além dos registradores compartilhados, cada p_i possui variáveis locais:

- r_i – número da rodada corrente de p_i;
- e_i – estimativa de decisão de p_i;
- $proposer_i$ – identidade do proponente corrente;
- $decision_i$ – valor decidido;
- aux_i – arranjo de registradores (usado como variável auxiliar para ler R);
- $prop_i$ – registrador (usado como variável auxiliar para ler um registrador compartilhado); possui os campos $\langle rd, vl, tg \rangle$;
- max_rd_i – número da maior rodada correntemente em R.

4.4.2 Algoritmo Genérico com n Conhecido

As primeiras versões do algoritmo consideram que o conjunto de participantes é conhecido. Os princípios que norteiam o algoritmo em todas as versões são os mesmos, de modo que não são apresentadas aqui as primeiras versões na íntegra, mas apenas uma descrição geral. As versões completas podem ser obtidas em (KHOURI; GREVE, 2013a, 2013b).

O algoritmo é composto das funções CONSENSUS e PROPOSITION. Todo p_i começa executando CONSENSUS (v_i), onde v_i é o seu valor inicial, e então invoca a função PROPOSITION. Na primeira versão, foram definidas duas funções: PROPOSITION$_{\diamondsuit\mathcal{S}}$ (e_i, r_i) e PROPOSITION$_{\Omega}$ (e_i, r_i), que são invocadas conforme o oráculo, onde e_i é a estimativa do processo p_i e r_i é a rodada corrente de p_i. Independente do detector, em PROPOSITION, o número da rodada é atualizado e um processo é definido como "proponente", o qual tenta impor uma proposta.

Se o detector é $\diamondsuit\mathcal{S}$, a cada invocação o número da rodada é incrementado de 1. O proponente, $proposer_i$, é escolhido em função do número da rodada e de n (mais precisamente, $proposer_i \leftarrow (r_i \bmod n) + 1$). O proponente então alterna entre leituras do arranjo de registradores e escritas em seu registrador $R[proposer_i]$ a fim de impor sua estimativa como proposta. Ao escrever uma estimativa, ele escreve, além do valor, o rótulo *est* em $R[proposer_i].tg$. Se ele escreve uma proposta, atualiza o rótulo para *pro* e retorna para CONSENSUS, a fim de tomar uma decisão.

Enquanto isso, os processos não proponentes aguardam para obter a proposta. Se em algum instante, um processo p_i suspeita de $proposer_i$, ele parte para uma nova rodada onde definirá um novo proponente. Se, por outro lado, obtém alguma proposta, retorna para CONSENSUS e tenta decidir. Por sua vez, se $proposer_i$ percebe que existe outro proponente concorrendo com ele que esteja em uma rodada mais adiantada, ele abandona a rodada, desistindo de sua posição de proponente e retorna para CONSENSUS. Ele divulga essa ocorrência escrevendo uma proposta inválida ($\perp$) em seu registrador. Assim os não proponentes que aguardam por ele, obtém a proposta (inválida), escrevem-na em seu registrador e também retornam para CONSENSUS.

Se o detector é Ω, o comportamento dos processos, proponentes ou não, será o mesmo descrito acima, exceto com relação à atualização do número da rodada e escolha do proponente. Nesse caso, o proponente escolhido é o líder definido pelo oráculo, o qual é o único a mudar de rodada. Além disso, cada rodada só é executada pelo proponente. Para garantir que isso ocorra, o número da primeira rodada para um proponente p_i é a sua própria identidade: $r_i = i$; e as subsequentes são dadas por $r_i \leftarrow r_i + n$.

Ao retornar para CONSENSUS, independente do oráculo, cada p_i lê o arranjo de registradores buscando por uma proposta. Então, se existirem propostas (pelo menos uma) em R, todas iguais a um valor v, e não existe qualquer proposta inválida ($= \perp$), p_i decide v. Caso contrário, parte para uma nova rodada.

Na segunda versão do algoritmo (KHOURI; GREVE, 2013b), foram eliminadas algumas ações que distinguiam PROPOSITION$_{\diamondsuit\mathcal{S}}$ (e_i, r_i) e PROPOSITION$_{\Omega}$ (e_i, r_i), no sentido de uniformizá-las, a ponto de fundi-las em uma única função PROPOSITION $(v_i, \mathcal{D})$, onde $\mathcal{D}$ é o detector instanciado. Adicionalmente, o algoritmo foi melhorado em termos de complexidade. Nesta versão, apenas o proponente escreve sua proposta no registrador

compartilhado, os demais processos só escrevem no registrador quando chegam a uma decisão.

4.4.3 Algoritmo Genérico com n Desconhecido – GENERICON

A última versão do algoritmo leva em conta a implementação da memória compartilhada num sistema distribuído dinâmico e admite a existência de uma operação $join()$ que prepara os processos no sistema a interagir com o recém-chegado e vice-e-versa. Um processo que deseja entrar na computação, requisita a operação $join()$ que se encarrega de lhe fornecer uma identidade i, e o estado dos registradores no sistema; e criar o seu registrador, $R[i]$, dando acesso de leitura aos demais participantes.

Algorithm 4 GENERICON $(v_i, \mathcal{D})$ process p_i

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

Seguindo o raciocínio das primeiras versões, o algoritmo GENERICON (junto com a função PROPOSITION) baseia-se na ideia de que um processo proponente tenta impor uma decisão, enquanto que não proponentes limitam-se a aguardar por uma decisão. Uma vez que n não é conhecido e pode variar a cada rodada, a forma utilizada para atualizar o número da rodada e definir o proponente foi adaptada a esse novo requisito.

O proponente, $proposer_i$, é definido em GENERICON $(v_i, \mathcal{D})$, que consiste de um laço infinito executado por todo p_i, onde v_i é o valor inicial de p_i e $\mathcal{D}$ é o detector utilizado. Se

$\mathcal{D} \in \Omega$, o proponente é o líder retornado pelo oráculo. Se $\mathcal{D} \in \diamondsuit\mathcal{S}$, 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 0 ($proposer_i = p_0$) 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 0. Assim, enquanto um valor não é decidido, todo processo tem a chance de ser proponente.

Algorithm 5 PROPOSITION $(v_i, \&r_i)$ process p_i

24: $e_i \leftarrow v_i$; $r_i \leftarrow r_i + 1$;
FASE 1:
25: **write** $(R[i], \langle r_i, e_i, est\rangle)$
26: **read** $(R[1..ns], aux_i[1..ns])$;
27: **if** $(\exists j : aux_i[j].tg = \text{dec})$ **then**
28: **write**$(R[i], \langle -, aux_i[j].vl, \text{dec}\rangle)$;
29: **return** $aux_i[j].vl$;
30: **else**
31: **if** $(\exists\ j \neq i : aux_i[j].rd \geq r_i)$ **then**
32: $max\_rd_i \leftarrow j : \forall k : aux_i[j].rd \geq aux_i[k].rd$;
33: **if** $(max\_rd > r_i)$ **then**
34: $r_i \leftarrow max\_rd$;
35: **write**$(R[i], \langle -, \perp, \text{dec}\ \rangle)$;
36: **return** $\perp$;
37: **else**
38: **if** $(\exists j : (aux_i[j].tg = \text{pro})$ **then**
39: $e_i \leftarrow (aux_i[j].vl{:}\ \forall j, k{:}aux_i[j].tg = aux_i[k].tg{=}\text{pro}, aux_i[j].rd \geq aux_i[k].rd)$;
FASE 2:
40: **write** $(R[i], \langle r_i, e_i, \text{pro}\rangle)$;
41: **read** $(R[1..ns], aux_i[1..ns])$;
42: **if** $(\exists\ j : aux_i[j].rd > r_i)$ **then**
43: $max\_rd_i \leftarrow j : \forall k : aux_i[j].rd \geq aux_i[k].rd$;
44: $r_i \leftarrow max\_rd$;
45: **write**$(R[i], \langle -, \perp, \text{dec}\ \rangle)$;
46: **return** $\perp$;
47: **else**
48: **if** $(\exists j : aux_i[j].tg = \text{dec})$ **then**
49: $e_i \leftarrow (aux_i[j].vl{:}\ \forall j, k{:}aux_i[j].tg = aux_i[k].tg{=}\text{dec}, aux_i[j].rd \geq aux_i[k].rd)$;
50: **write** $(R[i], \langle r_i, e_i, \text{dec}\rangle)$;
51: **return** e_i;

Um processo p_i só invoca PROPOSITION $(v_i, \&r_i)$ se ele se considera proponente (i.e., $proposer_i = i$), onde r_i é a rodada corrente de p_i. No início da execução de GENERICON, p_i define $proposer_i$ conforme o detector (linhas 3-4 ou 5-6) e então segue um dos caminhos: (1) Se $proposer_i = i$, p_i invoca PROPOSITION $(v_i, \&r_i)$ (linhas 7-8). Então p_i ajusta

sua estimativa corrente, e_i, para seu valor inicial, v_i, e incrementa o número da rodada (linha 24). 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 status de *decisão*. Isso acontece em duas fases quase idênticas – linhas 25-39 e 40-51.

Na fase 1, p_i começa escrevendo em $R[i]$ seus dados correntes, r_i e e_i, junto com o rótulo "est", que identifica o estágio de maturidade da informação (linha 25). Em seguida, p_i lê os registradores compartilhados (linha 26). Se p_i obtém alguma decisão em R, ele adota essa decisão como a sua e retorna para GENERICON (linhas 27-29), preservando o acordo. Se por outro lado, p_i percebe que há algum outro proponente em uma rodada maior ou igual à sua, ele desiste de tentar impor uma decisão e informa a todos sobre sua desistência, escrevendo um valor inválido ($\perp$) junto com o rótulo "dec" em $R[i]$ e retorna $\perp$ (linhas 31-36). Antes de retornar, porém, p_i atualiza seu número de rodada para o maior número de rodada em R. Outra verificação como esta da fase 1 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, ele altera sua estimativa para este valor (linhas 38-39) 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 40-46). Também desta vez, p_i pula para a rodada mais alta em R. Se, no entanto, ele persiste, verifica se há alguma decisão em R e, em caso positivo, altera sua estimativa para este valor; caso contrário, p_i mantém sua estimativa (linhas 48-49). Por fim, p_i decide e dissemina sua decisão escrevendo-a em $R[i]$, com o rótulo apropriado, "dec", retornando para GENERICON (linhas 50-51). Uma vez que um valor $v \neq \perp$ é retornado, p_i retorna v para a aplicação (linhas 8-9). 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 11-14 ou 15-18). No entanto, se p_i suspeita do proponente, abandona a espera e começa nova iteração. Se $\mathcal{D} \in \Omega$, a suspeita se dá quando Ω retorna uma identidade diferente do proponente corrente (linha 14). Se $\mathcal{D} \in \diamondsuit\mathcal{S}$, o proponente aparece na lista de suspeitos (linha 18)[1]. 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 19-20). Por fim, se p_i obtém uma decisão $v \neq \perp$, decide o mesmo valor, o qual retorna para a aplicação (linhas 21-23) .

4.5 PROVA DE CORRETUDE

Esta seção apresenta um esquema da prova de que o Algoritmo 4 – GENERICON $(v_i, \mathcal{D})$, com ajuda do Algoritmo 5 – PROPOSITION $(v_i, \&r_i)$, satisfaz às propriedades do Consenso. Todo processo p_i invoca GENERICON $(v_i, \mathcal{D})$, passando seu valor inicial v_i como argumento,

[1]Note que quando $proposer_i$ abandona a rodada, escreve um valor inválido com rótulo"dec" (linhas 35, 45), o que também libera p_i da espera.

o qual passa a ser sua estimativa (linha 1) para uma possível decisão (validade). Então p_i entra em um laço (linha 2) do qual só sairá após decidir um valor (a menos que falhe ou deixe o sistema). Numa iteração em que p_i é proponente, ele tenta decidir um valor que será copiado por todo p_j tal que $proposer_j = i$. Porém, como foi dito na Seção 4.4, é possível que mais de um processo considere-se proponente concomitantemente. 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/vivacidade).

Notação.

- É dito que "um processo p_i propõe um valor v" quando p_i escreve v em seu registrador junto com o rótulo "pro", isto é, executa a operação **write**($R[i]$, $\langle -, v,$ pro$\rangle$). Da mesma forma, considera-se que "p_i decide um valor v" quando executa a operação **write**($R[i]$, $\langle -, v,$ dec $\rangle$) e retorna o valor v.
- $\mathcal{D}$ denota um detector que pode ser da classe $\diamondsuit\mathcal{S}$ ou Ω.
- SM-$\mathcal{AS}[\Pi_{\mathsf{BC}}, f_c, \mathcal{D}]$ denota um sistema assíncrono de memória compartilhada, dinâmico (conforme o modelo de chegada infinita com concorrência limitada), sujeito a falhas por parada (crash), estendido com um detector Ω ou $\diamondsuit\mathcal{S}$.

Nas provas é utilizada a Proposição 2.2.1 transcrita a seguir.

Proposição 2.2.1 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.

Lema 4.1 (Validade)**.** *Em um sistema* SM-$\mathcal{AS}[\Pi_{\mathsf{BC}}, f_c, \mathcal{D}]$*, se algum processo p_i invoca* GENERICON$(-, \mathcal{D})$ *e decide v, então v é o valor inicial de algum processo.*

Prova:

Um processo p_i pode decidir um valor executando as linhas 22, 28 ou 50. Nas linhas 22 e 28, p_i decide um valor v que já foi decidido, obtido por ele em algum registrador (linhas 13 ou 17, 21-22; e linha 26, respectivamente). Então, é suficiente analisar o caso em que p_i decide na linha 50. Nesse caso, o valor decidido é o que está armazenado em e_i.

Todo $R[j].value$ é inicializado com $\perp$. Na linha 24, p_i atribui a e_i seu valor inicial v_i. Assim, na primeira vez que $p_i, \forall i$, escreve em $R[i].value$, escreve seu valor inicial (linha 25). Nas demais operações de escrita, o valor escrito por p_i é seu valor inicial (atribuído a e_i na linha 24) ou o valor obtido em algum $R[j]$ e atribuído a e_i (linhas 26, 38-40 ou 41,

48-50). Portanto, todo valor $v \neq \perp$ em $R[j].value, \forall j$, é o valor inicial de algum processo. Consequentemente, se p_i decide v, v é o valor inicial de algum processo. $_{Lema 4.1}\square$

Lema 4.2 (Acordo)**.** *Em um sistema* SM-$\mathcal{AS}[\Pi_{\mathsf{BC}}, f_c, \mathcal{D}]$*, se algum processo* p_i *invoca* GENERICON$(-, \mathcal{D})$ *e decide* v*, e outro processo* p_j *invoca* GENERICON$(-, \mathcal{D})$ *e decide* u*, então* $u = v$*.*

Prova:

Um processo p_i decide um valor v ao executar a linha 22, 28 ou 50. Se ele decide na linha 22 ou 28, v é um valor que foi decidido por algum p_j e obtido por p_i, respectivamente, na linha 13 ou 17; ou na linha 26, de modo que o Lema é satisfeito. Considere, então, que p_i decide na linha 50. 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. A análise contempla os seguintes casos:

Caso 1: $r_j > r_i$.
Considere as operações de escrita e leitura executadas por p_i nas linhas 40 e 41; e as operações de escrita e leitura executadas por p_j nas linhas 25 e 26. Pela Proposição 2.2.1, p_i obtém $R[j] = \langle r_j, -, - \rangle$ na linha 41, ou p_j obtém $R[i] = \langle r_i, e_i, \text{pro} \rangle$ (ou $R[i] = \langle r_i, e_i, \text{dec} \rangle$, se p_i já tiver executado a linha 50) na linha 26.

(A) Se p_i escreve $\langle -, v, \text{dec} \rangle$, na linha 50, é porque:

(1) não obteve $R[j] = \langle r_j, -, - \rangle$ na linha 41, ou teria desistido da rodada (linhas 42-46) sem atingir a linha 50;

(2) $e_i = v$ quando p_i executa a linha 40.

Portanto, quando p_j executa a linha 26, obtém (B) $R[i] = \langle r_i, v, \text{pro} \rangle$; ou (C) $R[i] = \langle r_i, v, \text{dec} \rangle$.

(B) Como, da hipótese, p_i e p_j são os únicos proponentes em I, e p_i é o primeiro a decidir, se p_j obtém $R[i] = \langle r_i, v, \text{pro} \rangle$, avalia os predicados das linhas 27 e 31 como *falso* e o da linha 38 com *verdadeiro*. Então atribui v a e_j na linha 39.

Em sua segunda leitura sobre R (linha 41), 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 42 como *falso* e, independente da avaliação do predicado da linha 48, executa a operação **write**$(R[j], \langle r_j, e_j, \text{dec} \rangle)$, com $e_j = v$, e retorna v para GENERICON (linhas 28-29) e para a aplicação (linhas 8-9). Portanto, o Lema é mantido.

(C) Se p_j obtém $R[i] = \langle r_i, v, \text{dec} \rangle$, na linha 26, avalia o predicado da linha 27 como *verdadeiro*, executa **write**$(R[j], \langle r_j, v, \text{dec} \rangle)$, e retorna v para GENERICON e para a aplicação ((linhas 28-29) e 8-9) de modo que o Lema é satisfeito.

Caso 2: $r_j = r_i$.

Considere as operações de escrita e leitura executadas nas linhas 25 e 26, respectivamente, por p_i e p_j. Pela Proposição 2.2.1, 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 50, é porque não obteve $R[j] = \langle r_j, -, -\rangle$ na leitura da linha 26, caso contrário, p_i teria desistido da rodada, contrariando a hipótese. Então, p_j obtém $r_i = r_j$ (linha 26) 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 40 e 41, executadas por p_j na rodada r_j, bem como as operações de escrita e leitura das linhas 25 e 26, executadas por p_i na rodada r_i, respectivamente. Pela Proposição 2.2.1, p_i obtém obtém $R[j] = \langle r_j, e_j,$ pro$\rangle$ (ou $R[j] = \langle r_j, e_j,$ dec$\rangle$, se p_j já tiver executado a linha 50); ou p_j obtém $R[i] = \langle r_i, e_i,$ est$\rangle$.

(A) Se p_j obtém $R[i] = \langle r_i, e_i,$est$\rangle$, avalia o predicado da linha 42 como verdadeiro e desiste da rodada, de modo que o Lema se mantém.

(B) Se, no entanto, p_j não obtém $R[i] = \langle r_i, e_i,$est$\rangle$, p_i obtém $R[j] = \langle r_j, e_j,$ pro$\rangle$ na linha 26. Seja $e_j = v$. Uma vez que p_i e p_j são os únicos proponentes, e p_i é o primeiro a decidir, p_i avalia os predicados das linhas 27 e 31 como *falso*. Na linha 38, p_i avalia o predicado como *verdadeiro*, de modo que atribui v a e_i na linha 39.

Na linha 41, p_i obtém, novamente, $R[j] = \langle r_j, v,$ pro$\rangle$. Mais uma vez, como p_i e p_j são os únicos proponentes, p_i avalia o predicados das linhas 42 e 48 como *falso* e decide $e_i = v$ na linha 50.

Por sua vez, se p_j não obtém $R[i] = \langle r_i, v,$ est$\rangle$ na linha 41, avalia o predicado das linhas 42 e 48 como *falso* e decide v na linha 50, de maneira que o lema é satisfeito.

*Lema*4.2□

Lema 4.3. *Em um sistema* SM-$\mathcal{AS}[\Pi_{\mathsf{BC}}, f_c, \mathcal{D}]$, *pelo menos um processo estável que invoca* PROPOSITION$(-, r_i)$ *decide um valor* v.

Prova:

Conforme explicado na Seção 4.4, 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 a propriedade assumida no modelo do sistema, de que pelo menos um processo permanece ativo para sempre, e conforme o oráculo instanciado, tem-se que:

(A) Se $\mathcal{D} \in \Omega$, 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, se p_i não decide antes, $proposer_i = s, \forall i$.

(B) Se $\mathcal{D} \in \Diamond\mathcal{S}$, 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, se p_i não decide antes, 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 34 ou 44), e a cada invocação de PROPOSITION, o número de rodada é incrementado de 1 (linha 24), após um tempo, p_s terá o número de rodada mais alto em R. Então, como p_s não é mais considerado suspeito, se outro processo não decide antes, p_s acaba decidindo v na linha 50.

$Lema 4.3$ □

Lema 4.4. *Em um sistema* SM-$\mathcal{AS}[\Pi_{\mathsf{BC}}, f_c, \mathcal{D}]$, *nenhum processo estável que invoca* GENERICON$(-, \mathcal{D})$ *permanece bloqueado indefinidamente.*

Prova:

O único trecho do algoritmo em que um processo p_i pode ficar bloqueado é no **repeat-until** das linhas 12-14 ou 16-18. Em qualquer caso, p_i fica aguardando que $proposer_i$ decida um valor. Para garantir que p_i não fica indefinidamente bloqueado, deve-se mostrar que pelo menos uma das condições é satisfeita:

Condição 1: $R[proposer_i].tg = \text{dec}$;
Condição 2: $proposer_i \neq leader()$ (se $\mathcal{D} \in \Omega$);
Condição 3: $proposer_i \in suspected_i$ (se $\mathcal{D} \in \diamondsuit\mathcal{S}$).

Todas as condições dependem do comportamento de $proposer_i$ e dos oráculos. Daí, tem-se que:

(A) Se $proposer_i$ decide v, escreve sua decisão em seu registrador (linhas 28 ou 50) e a *Condição 1* é satisfeita.

(B) Se $proposer_i$ falha ou deixa o sistema, então,

(B.1) se $\mathcal{D} \in \Omega$, 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.

(B.2) Se $\mathcal{D} \in \diamondsuit\mathcal{S}$, a propriedade *completude forte* garante que em algum momento $proposer_i \in suspected_i$ e a *Condição 3* é satisfeita.

(C) Se $proposer_i$ abandona a rodada sem decidir um valor, antes de retornar da função PROPOSITION, $proposer_i$ escreve $\langle -, \perp, \text{dec}\rangle$ em seu registrador, e então a *Condição 1* é satisfeita.

(D) Se $proposer_i$ não se considera proponente, não invoca a função PROPOSITION $(v_i, \&r_i)$. Mas, pelo Lema 4.3, 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. $Lema 4.4$ □

Lema 4.5 (Terminação)**.** *Em um sistema* SM-$\mathcal{AS}[\Pi_{\mathsf{BC}}, f_c, \mathcal{D}]$*, todo processo estável que invoca* GENERICON$(-, \mathcal{D})$ *decide um valor* v*.*

Prova:

Pelo Lema 4.4, nenhum processo fica bloqueado executando GENERICON. Pelo Lema 4.3, ao cabo de um tempo, pelo menos um processo estável p_s decide um valor e é tal que $proposer_i = s, \forall i$, decide v (ver prova do Lema 4.3). Então, após este instante, todo processo estável obtém o valor decidido (linhas 13 ou 17) e decide o mesmo valor (linhas 21-23). $_{Lema 4.5}\square$

Teorema 4.1. *Em um sistema* SM-$\mathcal{AS}[\Pi_{\mathsf{BC}}, f_c, \mathcal{D}]$*, o algoritmo* GENERICON$(v_i, \mathcal{D})$ *(com a ajuda do algoritmo* PROPOSITION$(v_i, \&r_i)$*) satisfaz as propriedades do Consenso (definidas na Seção 4.2.1).*

Prova: Segue diretamente dos Lemas 4.1, 4.2 e 4.5. $_{Teorema 4.1}\square$

Teorema 4.2. *O algoritmo* GENERICON$(v_i, \mathcal{D})$ *(juntamente com o algoritmo* PROPOSITION$(v_i, \&r_i)$*) é livre de espera.*

Prova:

O algoritmo GENERICON$(v_i, \mathcal{D})$ é correto, a despeito de $(n-1)$ falhas (Teorema 4.1). 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, pela primeira vez, p_i pode encontrar uma rodada superior à sua em R, o que o fará abandonar a execução de PROPOSITION e ajustar seu número de rodada para o maior em R (linhas 31-36 e 42-46).

Mas na próxima vez em que p_i invocar PROPOSITION, não encontrará mais qualquer rodada superior à sua, de modo que ele consegue atingir a linha 50, quando decide um valor.

Considere agora uma execução na qual menos de $(n-1)$ processos falham. Pelo Lema 4.3, ao menos um processo estável p_s decide um valor. Após uma decisão ser escrita em R, todo proponente em PROPOSITION decide o mesmo valor (linhas 27-29 ou linhas 41, 48-50). Assim, qualquer que seja o processo considerado proponente por p_i, se não falha antes, decide v. Então, p_i obtém a decisão (linhas 11-14 ou 15-18) e decide o mesmo valor (linhas 21-23).

É preciso mostrar então que p_i, ao cabo de um tempo, definirá um proponente que chegará a uma decisão. Pelas propriedades dos oráculos, tem-se que a partir de um instante t:

(1) Se $\mathcal{D} \in \diamondsuit\mathcal{S}$, pelo menos um processo estável, p_s, não será mais considerado suspeito. Então, seguindo o rodízio do coordenador rotativo, se p_i não decide antes, ao cabo de um tempo, $proposer_i = p_s$.

(2) Se $\mathcal{D} \in \Omega$, todo processo define o mesmo processo estável, p_s, como proponente.

Logo, ao cabo de um tempo, p_i decide um valor e o Teorema é satisfeito. $_{Teorema 4.2}\square$

4.6 RESILIÊNCIA E COMPLEXIDADE

O algoritmo apresentado é ótimo quanto à resiliência, uma vez que, conforme Teorema 4.2, tolera a falha de até $n-1$ processos. São utilizados n registradores regulares $1WMR$, um para cada processo que entra no sistema. Embora não tenham sido identificados trabalhos que estabeleçam o número mínimo de registradores requeridos para resolver o consenso num ambiente dinâmico de memória compartilhada, o limite n estabelecido em (LO; HADZILACOS, 1994) para algoritmos de consenso livres de espera em sistemas estáticos é um forte indício de que o algoritmo proposto aqui apresenta complexidade ótima. Entretanto, uma vez que o número de rodadas não é limitado, o tamanho dos registradores é ilimitado também.

Quanto à complexidade de passo, o algoritmo proposto apresenta o mesmo custo do fornecido em (GUERRAOUI; 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, e portanto, de interesse, é quando em tais sistemas, durante alguns períodos, a comunicação ocorre de maneira sincronizada. Nesses períodos, detectores não confiáveis podem ser bem precisos. Keidar e 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 (CRISTIAN; FETZER, 1999), isto é, não comete erros.

Considerando execuções bem-comportadas, 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 25, 40 e 50) e duas operações de leitura sobre o arranjo de registradores para decidir (linhas 26 e 41). Os demais processos obtém a decisão no registrador do proponente (linha 13 ou 17), o que pode ocorrer após uma única leitura.

4.7 DISCUSSÃO E TRABALHOS RELACIONADOS

O problema do consenso tem sido amplamente estudado e vários algoritmos propostos, a maioria deles para sistemas de passagem de mensagens (CHANDRA; TOUEG, 1996; LAMPORT, 1998; GUERRAOUI; RAYNAL, 2004). Para o modelo de memória compartilhada, foram identificados alguns artigos (GUERRAOUI; RAYNAL, 2007; DELPORTE-GALLET et al., 2004; LO; HADZILACOS, 1994).

Lo e Hadzilacos (1994) propõem dois algoritmos de consenso para um sistema assíncrono com o conjunto de participantes conhecido. O primeiro algoritmo considera que o sistema assíncrono é estendido com um detector de falhas da classe *forte* ($\mathcal{S}$), enquanto que o segundo utiliza um detector forte após um tempo ($\diamondsuit\mathcal{S}$). Eles usam *registradores atômicos*, e provam que são necessários pelo menos n registradores atômicos $1WMR$ para construir algoritmos de consenso *wait-free*, usando detectores de falhas *forte*, onde n é a cardinalidade do conjunto de participantes. O algoritmo aqui apresentado funciona corretamente com n registradores $1WMR$ mais fracos (regulares), 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.

Guerraoui e Raynal (2004) identificam a estrutura da informação de algoritmos de

consenso indulgentes, considerando detectores de falhas. Eles apresentam um algoritmo de consenso genérico que pode ser instanciado com um oráculo específico e mantém a mesma complexidade de acordo com alguns critérios. Diferentemente do proposto aqui, eles consideram um modelo onde os processos se comunicam por passagem de mensagens e o número máximo de falhas é $f < n/2$. Em outro artigo (GUERRAOUI; RAYNAL, 2007), eles apresentam um arcabouço que unifica uma família de algoritmos de consenso com base no detector Ω. O algoritmo pode ser ajustado para diferentes modelos de comunicação, mas em qualquer um dos modelos só consideram Ω. A complexidade de tempo (baseada nos acessos à memória compartilhada) é a mesma do algoritmo proposto aqui.

Delporte-Gallet *et al.* (2004) apresentam dois algoritmos para consenso em memória compartilhada considerando um sistema estendido com um detector Ω em que os processos são anônimos. Em um dos algoritmos, a memória compartilhada é composta de registradores atômicos multi-escritores-multi-leitores ($MWMR$) e o conjunto de participantes do sistema é ilimitado. O algoritmo usa dois registradores para cada rodada e o número de rodadas é ilimitado, assim, o número de registradores é ilimitado também. O outro algoritmo utiliza n registradores atômicos $1WMR$, mas considera um conjunto de participantes fixo.

Não foram identificados na literatura outras propostas de algoritmos para resolução do consenso em sistemas dinâmicos de memória compartilhada nem a definição de um limite inferior para a complexidade de espaço (número mínimo de registradores necessários) de tais algoritmos.

4.8 CONCLUSÃO

Neste capítulo foi apresentado um algoritmo genérico para resolver o consenso tolerante a falhas de parada em um sistema assíncrono de memória compartilhada, estendido com $\diamondsuit\mathcal{S}$ ou Ω. Mostrou-se que é possível construir um algoritmo genérico baseado em oráculo para o modelo de memória compartilhada, em que a cardinalidade do conjunto de participantes do sistema é desconhecida. 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 que se sabe, este é o primeiro protocolo de consenso genérico quanto ao oráculo para sistemas dinâmicos de memória compartilhada, que não exige o conhecimento de n. Considerando que o algoritmo pode ser instanciado com um oráculo da classe $\diamondsuit\mathcal{S}$ ou Ω (ou, ainda, processos distintos podem instanciar o algoritmo com oráculos distintos – $\diamondsuit\mathcal{S}$ ou Ω), portanto, o algoritmo é ótimo quanto ao sincronismo mínimo necessário para resolver o consenso. Também é ótimo no que diz respeito à resiliência, porque tolera qualquer número de falhas de processos $(n - 1)$.

Este resultado contribui para a criação de soluções práticas com foco em sistemas multinúcleo e sua evolução em direção a sistemas de *exascale*, bem como 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.

Um outro problema de interesse, fortemente relacionado ao consenso é a implementação de detectores de falhas ou de líderes que possam ser utilizados para estender sistemas assíncronos de modo a possibilitar a resolução do consenso através de algoritmos determinísticos. Neste sentido, foi investigado o problema da eleição de líder após um tempo e produzido um protocolo para implementação de um detector da classe Ω, o qual é apresentado no próximo capítulo. Assim como no problema do consenso, foi considerado um ambiente dinâmico de memória compartilhada em que o conjunto de participantes é desconhecido.

Capítulo

5

Um detector de líder é uma abstração poderosa para garantir a consistência e tolerância a falhas em sistemas distribuídos dinâmicos. Este capítulo apresenta um serviço de líder da classe Ω para o modelo de memória dinâmica compartilhada, sujeito a falhas de processos. Diferentemente das soluções conhecidas, ele não requer o conhecimento da cardinalidade dos participantes do sistema, de tal forma que os nós podem entrar e sair livremente do sistema.

ELEIÇÃO DE LÍDER APÓS UM TEMPO EM MEMÓRIA COMPARTILHADA COM BASE EM PADRÃO DE MENSAGENS

5.1 INTRODUÇÃO

No projeto de sistemas distribuídos confiáveis, camadas superiores de serviço como consenso, difusão atômica ou replicação de máquinas de estados, contam com um *serviço de eleição de líder após um tempo* para lidar com falhas de processos. Um detector de líder da classe Ω, funciona como um oráculo que, quando requisitado por um processo, fornece a identidade de um processo considerado correto no sistema.

Apesar de um oráculo Ω fornecer informações equivocadas durante um intervalo de tempo finito, após esse período de instabilidade, todo processo do sistema recebe informações consistentes do serviço, a saber, a identidade do mesmo processo correto; diz-se então que todo processo no sistema confia naquele eleito como *líder*. A classe Ω reúne as condições minimais para se resolver o consenso (CHANDRA; HADZILACOS; TOUEG, 1996), um serviço fundamental que fatora a necessidade de acordo no sistema.

Mas não há como implementar Ω em um sistema assíncrono pelos mesmos motivos atribuídos à impossibilidade do consenso. Na realidade, se fosse possível resolver o consenso em um sistema assíncrono com a ajuda de um detector de falhas que pudesse ser implementado em um sistema assíncrono, a impossibilidade do consenso seria uma contradição! (MOSTEFAOUI; RAYNAL; TRAVERS, 2004) apresentam uma prova direta de que é impossível implementar Ω em um sistema assíncrono sujeito a falhas por parada.

Neste capítulo é apresentado um algoritmo para Ω num sistema assíncrono sujeito a falhas por parada (*crash*), em que o conjunto de participantes é desconhecido e a comunicação se dá através de uma memória compartilhada. Dada a impossibilidade de se implementar Ω em tal ambiente, serão feitas algumas suposições adicionais a respeito do modelo do sistema.

Algumas abordagens voltadas para o modelo de passagem de mensagens foram propostas. Inicialmente, elas consideravam um modelo com fortes requisitos de sincronia e confiabilidade dos canais (CHANDRA; HADZILACOS; TOUEG, 1996; AGUILERA et al., 2001), mas, paulatinamente, foram sendo apresentadas soluções com requisitos cada vez mais fracos, considerando particularmente a existência de canais com requisitos temporais apenas para o líder (AGUILERA et al., 2004; MALKHI; OPREA; ZHOU, 2005; HUTLE et al., 2009). Entretanto, a totalidade de tais propostas é para o modelo clássico de sistemas distribuídos, cujo conjunto de processos é conhecido, sujeito a um limite superior para o número de falhas, f. Alguns poucos trabalhos existem para sistemas dinâmicos, cujo conjunto de processos é desconhecido e varia ao longo da execução; eles diferem no grau de conectividade e de conhecimento requerido do sistema (JIMÉNEZ; ARÉVALO; FERNÁNDEZ, 2006; ANTA; JIMÉNEZ; RAYNAL, 2006).

Para a comunicação por memória compartilhada, poucas são as propostas para Ω e a maioria delas considera sistemas estáticos (GUERRAOUI; RAYNAL, 2006; FERNÁNDEZ; JIMÉNEZ; RAYNAL, 2007; FERNÁNDEZ et al., 2010). Uma estratégia menos frequente, consiste em assumir um *padrão de mensagens* no sistema, ao invés de considerar limites para passos de processos ou transferência de mensagens. Protocolos baseados nesta abordagem não utilizam temporizadores e são por isso chamados *livres de tempo* (ou *time-free*). Nessa linha, alguns trabalhos surgiram para o modelo de passagem de mensagem (MOSTEFAOUI; MOURGAYA; RAYNAL, 2003; ARANTES et al., 2013).

Pelo que se sabe, até recentemente, não existia uma implementação de Ω para memória compartilhada assíncrona baseada em suposições livres de tempo. Em (KHOURI; GREVE, 2014), é proposta uma primeira solução nesse sentido. Entretanto, o protocolo ali proposto não é eficiente, pois exige que todos os processos continuem a escrever na memória para sempre, mesmo após a eleição do líder. Os resultados ali apresentados servem como prova de conceito de que a eleição livre de tempo é possível.

Neste capítulo, uma versão otimizada do algoritmo é apresentada que, além de ser livre de tempo, é mais eficiente quanto às escritas. No caso, ele reúne as condições necessárias para atingir escritas ótimas, que é quando, após a eleição, somente o líder precisa continuar a escrever no seu registrador, enquanto os demais precisarão apenas ler. Isso ocorre sempre que os demais processos conseguem obter o último valor atualizado do registrador do líder. Ou seja, se todos os demais processos acessam a memória do líder somente depois que ele a atualizou. Com essa estratégia, tal como em (AGUILERA et al., 2004; MALKHI; OPREA; ZHOU, 2005; HUTLE et al., 2009) que visavam optimalidade de sincronia, requerendo emissão de mensagens através de canais síncronos apenas pelo líder, avança-se no estado da arte da implementação de Ω para memória compartilhada, requerendo que somente o líder continue a escrever na memória.

O restante do capítulo está organizado como segue. A seção 5.2 descreve o modelo do sistema considerando a especificação do detector de líder Ω no contexto dinâmico.

Na seção 5.3 é apresentado o algoritmo para eleição de líder; na seção 5.4 as provas de sua corretude e na seção 5.5 sua complexidade. A seção 5.6 traz uma discussão sobre trabalhos relacionados e a seção 5.7 conclui o capítulo.

5.2 MODELO DO SISTEMA

O sistema é composto por um conjunto infinito de processos, que se comunicam através de uma memória compartilhada. Não são feitas suposições sobre a velocidade relativa dos processos, para dar passos ou acessar a memória, isto é, o sistema é assíncrono. Para facilitar a apresentação, assume-se a existência de um relógio global que não é acessível pelos processos, cuja faixa de ticks, $\mathcal{T}$, é o conjunto dos números naturais.

5.2.1 Processos

Neste capítulo, consideram-se as mesmas suposições relativas aos processos e à implementação da memória compartilhada feitas no capítulo 4. Em resumo, o modelo é o de chegada infinita, onde o conjunto de processos participantes, Π_{BC}, em cada execução r é limitado a n. Os processos podem falhar por parada e não existe um limite definido para o número máximo de processos que podem falhar.

Um processo p_i que deseja entrar no sistema, invoca uma operação $join()$ que se encarrega de lhe atribuir uma identidade e informar sobre o estado do sistema. Após o término de $join()$, todos os participantes têm acesso de leitura ao registrador $R[i]$, de p_i, o qual é considerado *ativo* até que falhe ou deixe o sistema.

Para sistemas distribuídos clássicos de passagem de mensagem, o número de processos no sistema, n, e o limite superior para o número de falhas, f, desempenham papel fundamental no projeto de algoritmos, conquanto juntos, eles representam um limite inferior sobre o número de processos corretos para que o algoritmo convirja. Em um sistema dinâmico, onde nem n nem f são conhecidos, uma estratégia adotada é abstrair o par (n, f) através de um parâmetro α, que representa esse número mínimo de processos estáveis no sistema. É bem sabido que com passagem de mensagens, o limite máximo para o número de processos faltosos é $n/2$, enquanto que com memória compartilhada, esse limite sobe para $n-1$ (LO; HADZILACOS, 1994). Considerando a possibilidade de $f \leq (n-1)$ processos falharem, tem-se $\alpha \geq (n-(n-1))$, então $\alpha \geq 1$.

De modo semelhante, no modelo de memória compartilhada ora proposto, assume-se a existência de um parâmetro α, que define o número mínimo de processos $p_s \in STABLE$. Portanto, uma condição de progresso provida pelo modelo do sistema é dada pela Propriedade 5.2.1.

Propriedade 5.2.1 (Limite Inferior de $STABLE$)**.** $|STABLE| \geq \alpha$.

5.2.2 Memória Compartilhada

A memória compartilhada consiste de arranjos de registradores atômicos multivalorados do tipo 1-*escritor-múltiplos-leitores* ($1WMR$) que se comportam corretamente (LAMPORT, 1986) (para rever as definições referentes a memória compartilhada, consulte as seções 2.2.1 e 2.7.1).

5.2.3 Especificação de Ω

O uso de abstrações como detectores de falhas ou de líder possibilita a solução de certos problemas insolúveis em sistemas assíncronos sujeitos a falhas (CHANDRA; HADZILACOS; TOUEG, 1996). Esses dispositivos são constituídos de módulos distribuídos que funcionam como oráculos que fornecem "dicas" sobre falhas de processos (ou líderes). Um serviço *detector de líder após um tempo* da classe Ω provê os processos com uma função $Leader()$ que quando invocada por um processo p_i, fornece a identidade de um processo p_j que o detector considera correto naquele instante. Após um tempo, essa identidade é única para todos os processos que invocam a função. Num contexto de sistema dinâmico, um detector Ω satisfaz à seguinte propriedade:

Propriedade 5.2.2 (Liderança após um tempo (*eventual leadership*))**.** *Existe um instante após o qual qualquer invocação de $Leader()$ por qualquer processo $p_i \in STABLE$ retorna a identidade do mesmo processo $p_l \in STABLE$.*

Para implementar Ω num sistema assíncrono, é preciso enriquecê-lo com suposições adicionais de sincronia que permitam a convergência. Em geral, essas suposições envolvem aspectos temporais, de modo que os protocolos utilizam temporizadores para garantir a terminação. Ao invés disso, aqui é introduzida uma propriedade comportamental que reflete um padrão relativo aos acessos feitos à memória compartilhada.

No modelo considerado, em que a memória é constituída de registradores $1WMR$, o processo p_i, proprietário de um registrador $R[i]$, pode manifestar seu progresso incrementando $R[i]$ periodicamente. Cada p_j pode então conferir se cada p_i permanece vivo, observando o crescimento de $R[i]$. A propriedade proposta considera esse acompanhamento. Informalmente, supõe-se que existe um processo *estável*, p_l, e um instante t, após o qual, as escritas de p_l em $R[l]$ estão sempre entre as primeiras α atualizações percebidas entre todos os registradores compartilhados, do ponto de vista de qualquer processo *estável* no sistema.

5.2.4 Propriedade do Sistema

Considere um sistema em que cada processo ativo p_i escreve/atualiza periodicamente seu registrador $R[i]$ enquanto executa uma rodada de operações de leitura sobre os registradores $R[j]$ dos processos p_j, a fim de obter os (novos) valores escritos. A rodada termina quando p_i tiver obtido um número "suficiente" de valores atualizados e, na sequência, p_i inicia uma nova rodada. Seja $U_i^{r_i}$ o conjunto dos primeiros α processos p_j cujos registradores foram encontrados atualizados por p_i na rodada r_i. A Propriedade 5.2.3 impõe a existência de um processo p_l que, a partir de um instante t, está presente em $U_i^{r_i}$, para todo p_i estável, $\forall r_i$.

Propriedade 5.2.3 (Padrão de Acesso à Memória – PAM)**.** *Existe um instante $t \in \mathcal{T}$ e um processo $p_l \in STABLE$ tais que, $\forall t' \in \mathcal{T}, t' \geq t$, $\forall p_i \in STABLE, \forall r_i : p_l \in U_i^{r_i}$.*

Em última análise, a Propriedade 5.2.3 reflete alguma regularidade no comportamento do sistema, mesmo que ele se comporte de maneira assíncrona por um período arbitrário

de tempo. É como se a partir de t, a comunicação entre cada processo p_i e p_l estivesse entre as (α) mais rápidas quando se considera a comunicação entre p_i e cada p_j.

Nesse contexto, considera-se *bem comportada* uma execução em que a partir do instante t, em que PAM é satisfeita, p_l é tal que $R[l]$ é sempre o primeiro a ser encontrado atualizado numa rodada de leituras sobre os registradores.

5.3 O ALGORITMO LEADER_ELECTION()

O algoritmo é composto por três tarefas executadas em paralelo por todos os processos ativos. Um processo p_i define como seu líder o processo que ele considera "menos suspeito". Essa ideia de suspeição é expressa através de contadores indexados. Cada p_i possui registradores compartilhados $Punishments[i][j]$ que ele incrementa sempre que suspeita de p_j. Desse modo, em cada instante t, o estado de $Punishments[i][j]$ representa o número de vezes que p_i suspeitou (e puniu) p_j. Para não ser considerado suspeito, um processo p_j permanece incrementando seu registrador compartilhado $Alive[j]$. Por outro lado, um processo p_i considera p_j suspeito se ao ler $R[j]$ obtém o mesmo valor obtido desde sua última leitura.

O nível de suspeição de um processo p_i com relação a um processo p_j será definido pela soma dos contadores: $soma_i[j] = \sum_{\forall k} Punishments[k][j]$. O processo escolhido como líder será o p_j para o qual $soma_i[j]$ seja a menor. Se houver mais de uma soma com valor igual ao menor, o desempate será definido pela menor identidade de processo, isto é, será considerada a ordem lexicográfica:

$$MinLex(soma_i[x], soma_i[y]) = \begin{cases} x, & \text{se } soma_i[x] < soma_i[y] \text{ ou} \\ & \quad soma_i[x] = soma_i[y] \wedge x < y \\ y, & \text{caso contrario.} \end{cases}$$

Quando um processo p_r deseja entrar no sistema, invoca uma operação $join()$ que provê um estado consistente de todos os registradores para p_r; inicializa seus registradores $Alive[r]$ e $Punishments[r][i]$; e aloca espaço para seus arranjos dinâmicos. Os demais processos p_i, ao receberem uma solicitação de $join()$ da parte de p_r, criam seus registradores $Punishments[i][r]$ devidamente inicializados; e alocam espaço em seus arranjos dinâmicos locais para acomodar p_r. Apenas após o retorno de $join()$ um processo p_r estará apto a participar da computação e será considerado pelos demais processos no sistema como um candidato a líder (incluindo os registradores correspondentes na escolha).

Para impedir que um líder estável seja demovido por um recém-chegado, cada registrador $Punishments[i][r]$ é inicializado com o valor do contador de punições de p_i atribuído a p_l acrescido de 1: $Punishments[i][r] \leftarrow Punishments[i][l] + 1$; cada registrador $Punishments[r][i]$, $i \neq r$, é inicializado com 0; e $Punishments[r][r]$ é inicializado com 1. Dessa forma, o nível de suspeição inicial do recém-chegado ($\sum_{\forall i} Punishments[i][r]$) será maior que o do líder corrente.

Versão preliminar do algoritmo

Numa versão preliminar do algoritmo, publicada em (KHOURI; GREVE, 2014), todo processo p_i informa intermitentemente aos demais participantes que ele está vivo com

o fim de evitar falsas suspeitas a seu respeito. Ele faz isso em Task 1: Liveliness, incrementando seu registrador compartilhado $Alive[i]$. Por outro lado, todo p_i permanece lendo cada $Alive[j]$ para verificar o progresso de p_j. Ele se mantém fazendo essas leituras até que encontre, pelo menos, α $Alive[j]$ cujos valores foram atualizados entre duas leituras. Se entre duas leituras em algum $Alive[j]$ seu valor não é modificado, p_i pune p_j incrementando $Punishments[i][j]$.

O algoritmo funciona corretamente, isto é, satisfaz à propriedade de liderança após um tempo, mas apresenta a desvantagem de que todo p_i permanece escrevendo na memória compartilhada para sempre. Com o fim de otimizar as escritas, uma nova versão do algoritmo foi desenvolvida, a qual é apresentada a seguir.

Versão otimizada

O algoritmo é executado em paralelo por todos os processos ativos. Com o intuito de informar aos demais processos que está ativo, apenas o processo p_x que se considera líder mantém-se incrementando um contador num registrador compartilhado $Alive[x]$. Por outro lado, periodicamente, cada p_i que considera p_x como líder, consulta o valor em $Alive[x]$ para saber se p_x progrediu. Se p_i não consegue obter um valor atualizado, antes de suspeitar de p_x, inicia uma rodada de leituras sobre cada $Alive[j]$ e espera encontrar pelo menos α registradores atualizados. Se $Alive[x]$ está entre os α primeiros encontrados atualizados, não é considerado suspeito, caso contrário, p_i suspeita de p_x e lhe aplica uma punição, incrementando $Punishments[i][x]$.

O critério para a escolha do líder é o mesmo utilizado na versão preliminar. Como os processos compartilham a mesma visão de $Punishments$, p_i escolhe como líder o processo p_j para o qual $soma_i[j] = \sum_{\forall k} Punishments[k][j]$ é mínima, considerando a ordem lexicográfica para desempate.

5.3.1 Registradores Compartilhados e Variáveis Locais

No algoritmo são utilizados os seguintes registradores compartilhados:

- $Alive[i]$: contador que indica o progresso de p_i; inicializado com 0.
- $Punishments[i][j]$: contador de p_i para o número de punições aplicadas por p_i a p_j. No início, $Punishments[i][j] = 0, \forall i, j$. Quando um processo p_r entra, p_i inicializa $Punishments[i][r]$ com $Punishments[i][leader_i] + 1$, enquanto que $Punishments[r][i]$, $i \neq r$, é inicializado com 0; e $Punishments[r][r]$ é inicializado com 1.

Um processo p_i também utiliza as seguintes variáveis locais:

- $leader_i$ e ld_i – identidade do líder corrente em Task 1 e Task 2, respectivamente.
- $alive_i$ – controla o progresso de p_i cujo valor é escrito em $Alive[i]$. Inicializado com 0.
- $last\_leader_i$ e $last\_ld_i$ – identidade do líder da iteração anterior em Task 1 e Task 2, respectivamente.

- $last\_alive\_leader_i$ – último valor lido em $Alive[leader_i]$.
- $alive_i^j$ – variável auxiliar usada para leitura de $Alive[j]$.
- $last\_alive_i$ – vetor dinâmico que armazena o último valor lido em $Alive$. Cada entrada é inicializada com 0.
- $soma_i$ – vetor dinâmico que armazena a soma dos contadores de punições de cada p_j.
- $updated_i$ – conjunto para controle dos processos que têm seu progresso verificado.
- pun_i – variável auxiliar usada para leitura de $Punishments[i][j]$.

Notação:

- Diz-se que "p_i considera p_x como líder", se $leader_i = x$ (em TASK 1) ou $ld_i = x$ (em TASK 2).
- Diz-se que "p_x é demovido da posição de líder", se num instante qualquer, $leader_i = x$ ($ld_i = x$) e na próxima atribuição de valor, $leader_i$ (ld_i) recebe $y \neq x$.
- Diz-se que "p_i pune p_j" quando p_i incrementa seu registrador $Punishments[i][j]$.
- "Atualizar um registrador", significa escrever no registrador seu valor corrente incrementado de 1.
- Um registrador $Alive[j]$ é considerado "atualizado" por p_i se ele obtém numa leitura um valor maior que o obtido na leitura imediatamente anterior.

5.3.2 Entrando no Sistema

Para um processo p_r entrar no sistema, ele invoca uma operação $join()$ que lhe provê uma identidade e um estado consistente dos registradores; inicializa seus registradores compartilhados; e aloca espaço para seus arranjos dinâmicos locais. Os demais processos p_i, ao receberem uma requisição de $join()$, criam seus registradores $Punishments[i][r]$ devidamente inicializados com $Punishments[i][leader_i] + 1$ e alocam espaço nos arranjos dinâmicos locais para acomodar p_r. Apenas após o retorno de $join()$, um processo p_r estará apto a participar da computação.

5.3.3 Descrição do Algoritmo

Para entrar no sistema, p_i invoca a operação $join()$ e ao retornar inicia três tarefas em paralelo.

TASK 0: INIT-REGS – Ao perceber uma requisição de $join()$ de algum p_j, p_i inicializa o registrador $Punishments[i][j]$ com $Punishments[i][Leader()] + 1$ (linha 3).

Algorithm 6 LEADER_ELECTION() - Part 1

```
    Initialization
 1: join();
 2: start Task 0, Task 1, Task 2;
    Task 0: Init-Regs
    upon request join() from p_j
 3: write(Punishments[i][j], Punishments[i][Leader()] + 1);
    Task 1: Liveliness
 4: loop
 5:     leader_i ← Leader();
 6:     if (i = leader_i) then
 7:         alive_i ← alive_i + 1;
 8:         write (Alive[i], alive_i);
 9:         if (last_leader_i ≠ leader_i) then;
10:             last_leader_i ← leader_i;
11:     else
12:         read(Alive[leader_i], alive_leader_i);
13:         if (last_leader_i = leader_i) then
14:             if (last_alive_leader_i = alive_leader_i) then
15:                 alive_i ← alive_i + 1;
16:                 write (Alive[i], alive_i);
17:         else
18:             last_leader_i ← leader_i;
19:             last_alive_leader_i ← alive_leader_i;
```

Task 1: Liveliness – Nesta tarefa, quando um processo p_i considera-se líder, mantém-se atualizando $Alive[i]$, enquanto cada p_j verifica se $Alive[i]$ está atualizado. A tarefa consiste de um laço infinito no qual, inicialmente, cada processo p_i invoca a função $Leader()$ para definir $leader_i$ (linha 5). Para fazer certas verificações na iteração seguinte, p_i mantém a identidade do líder e de seu respectivo registrador $Alive$ em variáveis locais: $last\_leader_i$ – linhas 10 e 18; e $last\_alive\_leader_i$ – linha 19. Enquanto p_i considera-se líder ($leader_i = i$), mantém-se atualizando $Alive[i]$ (linhas 6-8).

Um p_i que não se considera líder ($leader_i \neq i$), mantém-se verificando se $leader_i$ está vivo. Para isso, p_i lê $Alive[leader_i]$ (linha 12) e verifica se o líder corrente é o mesmo da iteração anterior (linha 13). Se isso ocorre e $Alive[leader_i]$ não foi atualizado (linha 14), p_i atualiza seu $Alive[i]$ (linhas 15-16). Senão, p_i apenas guarda a identidade do líder e o valor de seu registrador para posteriores verificações (linhas 18 e 19). Isso significa que é possível que apenas o líder eleito permaneça escrevendo em $Alive$ pra sempre.

Task 2: Punishment – Aqui também são utilizadas variáveis locais, $last\_alive_i[j]$, para guardar o valor lido em $Alive[j]$ e possibilitar a verificação do progresso de p_j (linhas 27, 34). O objetivo desta tarefa é punir processos suspeitos, mas isso só ocorre se $leader_i$ é um dos suspeitos. Então, no início do laço, p_i define um líder, ld_i, invocando $Leader()$ (linha 21).

Nenhum processo suspeita de si mesmo. Então, enquanto p_i se considera líder, apenas invoca *Leader*() (linhas 21-22). Já um processo $p_i \neq ld_i$, inicializa o conjunto $updated_i$ com sua própria identidade (linha 23) e entra num laço **while** (linha 24) onde permanece até que uma das condições seja satisfeita – *Condição 1*: $ld_i \in updated_i$, o que significa que $Alive[ld_i]$ foi atualizado; ou *Condição 2*: $\mid updated_i \mid \geq \alpha$, que implica que pelo menos α processos p_j atualizaram $Alive[j]$. Como o algoritmo foi desenhado para sistemas assíncronos estendidos com as Propriedades 5.2.1 e 5.2.3, essas condições garantem que, ao cabo de um tempo, algum processo p_l sempre estará em $updated_i, \forall i$.

Algorithm 7 LEADER_ELECTION() - Part 2

TASK 2: PUNISHMENT

```
20: loop
21:     ld_i ← Leader();
22:     if (ld_i ≠ i) then
23:         updated_i ← {i};
24:         while ((ld_i ∉ updated_i) ∧ (|updated_i| < α)) do
25:             read (Alive[ld_i], alive_ld_i);
26:             if (alive_ld_i ≠ last_alive_i[ld_i]) then
27:                 last_alive_i[ld_i] ← alive_ld_i;
28:                 updated_i ← updated_i ∪ {ld_i};
29:             else if (ld_i ∉ updated_i) then
30:                 for all (j) do
31:                     if (j ∉ updated_i) then
32:                         read (Alive[j], alive_i^j);
33:                         if (alive_i^j ≠ last_alive_i[j]) then
34:                             last_alive_i[j] ← alive_i^j;
35:                             updated_i ← updated_i ∪ j;
36:             if ((| updated_i |< α) ∧ (ld_i ∉ updated_i)) then
37:                 if (leader_i ≠ ld_i) then ld_i ← leader_i;
38:         if (ld_i ∉ updated_i) then
39:             for (j ∉ updated_i) do
40:                 pun_i[j] ← pun_i[j] + 1;
41:                 write(Punishments[i][j], pun_i[j]);
```

Leader()

```
42: for all (j) do
43:     soma_i[j] ← 0;
44:     for all (k) do
45:         read(Punishments[k][j], pun1_i);
46:         soma_i[j] ← soma_i[j] + pun1_i;
47: leader_i ← MinLex(soma_i[j]);
48: return leader_i;
```

No **while**, p_i lê $Alive[ld_i]$ (linha 25). Se obtém um valor atualizado, guarda-o em $last\_alive_i[ld_i]$ e inclui ld_i em $updated_i$ (linhas 26-28). Dessa forma, a *Condição 1* é

imediatamente satisfeita. Então, p_i sai do laço, avalia o predicado da linha 38: *falso* e não pune qualquer p_j. Se, no entanto, p_i não obtém um valor atualizado no registrador do líder, entra num laço **for** (linhas 30-35) onde lê cada $Alive[j]$ e, caso obtenha um valor atualizado, inclui j em $updated_i$. Após o **for**, se a *Condição 1* é satisfeita, p_i sai do **while** e vai para a próxima iteração do laço (**loop**). Se a *Condição 2* é satisfeita e $ld_i \notin updated_i$, p_i pune cada $p_j \notin updated_i$ e vai para a próxima iteração do laço (**loop**) (linhas 38-41).

Para evitar que um processo permaneça bloqueado no **while**, ao sair do **for**, p_i verifica se o líder que ele obteve na última invocação a $Leader()$ em Task 1 – $leader_i$, é o mesmo sendo considerado em Task 2 – ld_i. Se não for, ele altera ld_i para receber $leader_i$ (linhas 36-37). Quando a Propriedade 5.2.3 se verifica, algum p_l eleito como líder por todo p_i estará sempre entre os α primeiros atualizados e a *Condição 1* é satisfeita. Além disso, p_l não será mais punido, portanto nenhuma escrita mais será efetuada sobre $Punishments$ (já que qualquer processo p_j só é punido se o líder também for).

A função $Leader()$ – Retorna uma identidade de processo que, do ponto de vista de p_i, é correto e considerado líder por ele. Para essa escolha, a soma das punições atribuídas a cada p_j é acumulada em $soma_i[j]$ (linhas 42-46). É escolhido como líder o processo com menor soma, utilizando a identidade como desempate (linhas 47-48).

5.4 CORRETUDE DO ALGORITMO

Esta seção apresenta uma prova de que os processos estáveis elegem um único processo estável – o líder, ao cabo de um tempo. Os Lemas propostos contribuem para a prova do Teorema que estabelece o cumprimento da propriedade de *liderança após um tempo*.

Num sistema distribuído com memória compartilhada, o comportamento de cada processo p_i depende do estado da memória que é observado por ele. No algoritmo proposto, as ações de p_i, enquanto executa Task 1: Liveliness, afetam diretamente o comportamento de um processo p_j, e do próprio p_i, executando Task 2: Punishment. O valor retornado pela função $Leader()$, por sua vez, depende do estado visto nos registradores $Punishments$, o qual é alterado pelos processos executando Task 2. Dessa forma, as provas do Lemas precisam recorrer, simultaneamente, às duas tarefas e à função.

Adicionalmente, devido a falhas de processos, o dinamismo e assincronia do modelo de sistema considerado, e ainda propriedades dos registradores atômicos, num mesmo instante, os processos p_i e p_j podem tomar decisões diferentes com base em suas respectivas visões da memória. Por exemplo, suponha que p_i e p_j invocam a função $Leader()$ (linha 5) num instante t, e recebem o retorno num instante t'. Suponha também que enquanto as operações de leitura são executadas por p_i e p_j sobre cada $Punishments[-][-]$ (linhas 42-46), operações de escrita são executadas concorrentemente sobre tais registradores. Dessa forma, a leitura de p_i sobre algum $Punishments[a][b]$ pode retornar um valor distinto daquele que é visto por p_j. Consequentemente, no instante t', é possível que $leader_i \neq leader_j$. Pode acontecer até mesmo de num instante t'', após a invocação de $Leader()$ por p_i, nas linhas 5 e 21, $leader_i \neq ld_i$.

Portanto, para garantir que ao cabo de um tempo, $leader_i = ld_i = leader_j = ld_j = l, \forall i, j$, é suficiente mostrar que a partir de um instante t, o estado de $Punishments$

permanece inalterado.

Lema 5.1. *Existe um instante t e um processo $p_l \in STABLE$ tais que após t, p_l não será jamais punido por qualquer processo p_i.*

Prova.

As punições ocorrem em TASK 2: PUNISHMENT (linhas 38-41). Se $p_i \in STABLE$, de acordo com a Propriedade 5.2.3, existe um processo p_l e um instante t', tais que em toda rodada de leituras sobre os registradores $Alive[j]$ (linhas 30-35), iniciada por p_i após t', $Alive[l]$ estará entre os α primeiros a serem encontrados atualizados com relação à última leitura feita por p_i. Assim, p_l será incluído em $updated_i$ (linhas 26-28 ou 33-35) antes que p_i saia do **while** (linha 24). Consequentemente, p_i não punirá p_l (linhas 38-41).

A Figura 5.1 ilustra o comportamento dos processos em torno do instante t' em que a Propriedade 5.2.3 é satisfeita. Considere as últimas rodadas de leituras sobre os registradores $Alive$, r_a, r_b e r_c, iniciadas antes de t', por cada p_i. Após cada rodada, se o predicado da linha 38 é avaliado como verdadeiro, punições são aplicadas em todo $p_j \notin updated_i$, ou seja, operações de escrita são executadas em $Punishments$ (linha 41). Seja $t \geq t'$ o momento em que a última destas operações de gravação, se houver alguma, é concluída. Depois de t, a avaliação do predicado da linha 38 levará em conta o conjunto $updated_i$ construído a partir das rodadas de leituras sobre $Alive$, r'_a, r'_b, r'_c, iniciadas após t'. Então, p_l estará em $updated_i, \forall i$, de modo que pode-se dizer que depois de t, nenhum p_i punirá p_l.

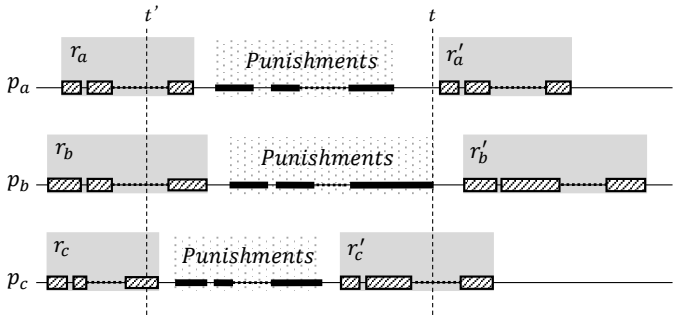

Figura 5.1 Cenário do sistema no instante t' em que a Propriedade 5.2.3 começa a ser satisfeita

Considere agora um processo p_r que entra no sistema em algum instante após t'. Como ocorre com qualquer outro processo, em toda rodada de leituras sobre $Alive$ executada por p_r, ele obtém $Alive[l]$ atualizado, de modo que, ao final de cada rodada, $p_l \in updated_r$, impedindo p_r de punir p_l.

Um processo p_i que falha ou sai do sistema antes do instante t, para de dar passos e, portanto, não pune p_l a partir de t.

Lema 5.2. *Existe um instante t e um processo $p_l \in STABLE$ tais que $\forall t' \geq t$, $\forall i$, $leader_i = ld_i = p_l$.*

Prova.

Para definir um processo p_x como líder, p_i invoca a função $Leader()$, que deve obter um estado de $Punishments$ tal que $soma_i[x]$ seja mínima, isto é, $MinLex(\sum_{\forall k} Punishments[k][j]) = x$. Portanto, uma vez que p_i define p_x como líder em um instante t, outro líder p_y só será definido em um instante $t' > t$, se seu ponto de vista do estado de $Punishments$ é mudado, de modo que $MinLex(\sum_{\forall k} Punishments[k][j]) = y \neq x$. Em outras palavras, um processo p_x é definido como líder por p_i, se a partir do ponto de vista de p_i ele é o menos punido entre todos; e p_x somente deixará de ser considerado líder por p_i se for punido (por qualquer processo) o suficiente para perder esta posição. Em ambos os casos, considere-se "menos punido" o processo retornado por $MinLex$.

Em Task 1, um processo p_x que se considera líder mantém-se atualizando seu registrador $Alive[x]$ (linhas 6-8). Um processo p_i que não se considera como líder, continua verificando se $Alive[leader_i]$ está atualizado (linha 12). Quando isso não ocorre, p_i atualiza seu próprio $Alive[i]$.

Em Task 2, um processo p_j é punido por p_i, se p_i não obtém $Alive[ld_i]$ nem $Alive[j]$ atualizado ao executar o **while** (linhas 24-37).

Considere então que $p_x \in STABLE$ considera-se como líder no instante t, ou seja, $leader_x = x$. Assim, ele permanece atualizando $Alive[x]$ (linhas 6-8). Para que p_x não seja punido por qualquer processo, é necessário que:

(1) todo p_i que considera p_x como líder, obtenha $Alive[x]$ atualizado antes de sair do **while**. Assim, p_i inclui x em $updated_i$ (linhas 25-28 ou 32-35) e quando ele deixa o **while**, avalia o predicado da linha 38 como falso e não pune qualquer processo; ou

(2) todo p_j que considera $p_y, y \neq x$, como líder,

  (2.1) obtenha $Alive[y]$ atualizado durante a execução do **while** (linhas 24-37). Assim, como em (1), cada p_j permanece sem punir qualquer processo; ou

  (2.2) se ele não obtém $Alive[y]$ atualizado durante o **while** da linha 24, obtém $Alive[x]$ atualizado e inclui x em $updated_j$ (linhas 30-35). Como p_j só vai punir os processos que não estão em $updated_j$, p_x não será punido.

Portanto, mesmo que p_x seja estável, pode ser punido, e a partir de uma determinada quantidade de punições, pode perder a posição de liderança. Isto pode ocorrer várias vezes, de modo que diferentes processos podem ser considerados líderes e demovidos desta posição. Mas, pelo Lema 5.1, existe um processo $p_l \in STABLE$ que a partir de um instante t', quando a Propriedade 5.2.3 é satisfeita, não é mais punido. Então, se todo processo $p_x \neq p_l$ considerado líder permanece sendo punido, em algum instante $t > t'$, p_l, que não é mais punido, será tal que $soma_i(l)$ será mínima, $\forall i$. Então, ele será eleito líder por todo p_i e não será mais demovido desta posição. $_{Lema 5.2}\square$

Lema 5.3. *Nenhum processo p_i fica permanentemente bloqueado executando o Algoritmo LEADER_ELECTION().*

Prova.

O único trecho do algoritmo em que um processo pode ficar bloqueado é o laço **while** em Task 2 (linhas 24-37), o qual só é executado por processos que não se consideram líderes (linha 22). Para mostrar que um processo não fica bloqueado no laço, deve-se mostrar que pelo menos uma das condições é satisfeita: *Condição 1*: $ld_i \in updated_i$; ou *Condição 2*: $\mid updated_i \mid \geq \alpha$ (linha 24).

Um processo p_j é incluído no conjunto $updated_i$ sempre que p_i lê $Alive[j]$ e percebe que seu valor foi atualizado desde a última leitura (linhas 25-28 ou 30-35). Assim, ambas as condições de saída do laço dependem da atualização dos registradores *Alive*, o que ocorre em Task 1 (linhas 8 e 16). Portanto, deve-se considerar o comportamento dos processos nas duas tarefas em paralelo.

Em Task 2, um processo p_i que se considera líder, permanece apenas invocando $Leader()$ (linhas 21-22). Já um processo $p_i \neq ld_i$, inicializa o conjunto $updated_i$ com sua própria identidade (linha 23) e entra no laço **while** (linha 24) onde permanece até que uma das condições seja satisfeita – *Condição 1*: $ld_i \in updated_i$, o que significa que $Alive[ld_i]$ foi atualizado; ou *Condição 2*: $\mid updated_i \mid \geq \alpha$, que implica que pelo menos α processos p_j atualizaram $Alive[j]$. As Propriedades 5.2.1 e 5.2.3 garantem que, ao cabo de um tempo, algum processo p_l, eleito líder, sempre estará em $updated_i, \forall i$, de modo que a *Condição 1* será sempre satisfeita.

Quando um processo p_x considera-se líder, ele atualiza seu registrador $Alive[x]$ periodicamente em Task 1 (linhas 6-8). Se p_x falhar em um instante t, a partir deste momento, as leituras que um processo p_i, que considera p_x como líder, executa sobre $Alive[x]$ irão sempre obter o mesmo valor (portanto, desatualizado). Neste caso, p_i atualiza seu próprio $Alive[i]$ (linhas 12-16). Mesmo se p_x é estável e atualiza $Alive[x]$, a leitura realizada por p_i pode não obter o valor atualizado devido à assincronia no sistema. Nesta situação, p_i também atualiza $Alive[i]$.

Em Task 2, p_i utiliza as informações obtidas nos registradores *Alive* para sair do laço **while** (linhas 24-37). Se ele obtém um valor atualizado em $Alive[x]$ (linha 25), p_i inclui p_x em $updated_i$ (linhas 26-28), de modo que a *Condição 1* é satisfeita. Caso contrário (linha 29), p_i tenta obter valores atualizados em pelo menos α registradores $Alive[j]$ (linhas 30-35). Para cada $Alive[j]$ que é encontrado atualizado, p_i inclui j em $updated_i$. Quando a cardinalidade de $updated_i$ atinge pelo menos α, a *Condição 2* é satisfeita.

Considere os seguintes casos possíveis em que um processo estável p_i que considera p_x como líder executa o **while** em Task 2.

Caso 1: p_x é estável e considera-se líder.

Então, p_x atualiza $Alive[x]$ (linha 8). Ao cabo de um tempo, se a *Condição 2* não é satisfeita antes, p_i obtém $Alive[x]$ atualizado (linha 25 ou 32) e a *Condição 1* é satisfeita.

Caso 2: p_x é faltoso e pelo menos α processos p_j o consideram como líder ($leader_j = x$) em Task 1 (linha 5).

Em um instante t, p_x falha e para de atualizar $Alive[x]$ (linha 8). Como consequência, em Task 1, a partir de um instante $t' > t$, toda leitura que p_j realiza em $Alive[x]$ (linha 12),

recebe um valor desatualizado. Neste caso, p_j atualiza seu próprio $Alive[j]$ (linhas 14-16). Em Task 2, p_i inclui em $updated_i$ cada p_j cujo registrador é encontrado atualizado (linhas 30-35). Portanto, depois de um tempo, devido à Propriedade 5.2.1, a *Condição 2* é satisfeita.

Caso 3: p_x é faltoso e menos de α processos estáveis p_j o consideram como líder ($leader_j = x$) em Task 1 (linha 5).

Em um instante t, p_x falha e para de atualizar $Alive[x]$ (linha 8). Neste cenário, p_i não inclui p_x em $updated_i$ (linhas 28 ou 35), de modo que a *Condição 1* não é satisfeita. Para garantir que p_i não fica bloqueado para sempre no **while**, deve-se mostrar que

(i) pelo menos α p_h atualizam $Alive[h]$ (linhas 8 ou 16) – quando essas atualizações são vistas por p_i em Task 2 (linha 32), p_i inclui todo p_h em $updated_i$ e a *Condição 2* é satisfeita; ou

(ii) p_i muda de líder; o que pode acontecer várias vezes até que uma das condições de saída do **while** seja atendida. A seguir explica-se porque p_i pode mudar de líder.

Uma vez que $|STABLE| \geq \alpha$ (Propriedade 5.2.1), e menos de α processos estáveis p_j consideram p_x como líder, há algum processo p_k tal que $leader_k = y, y \neq x$. Isto é devido à assincronia entre a escrita e leitura nos registradores $Punishments$.

Tome como exemplo o caso hipotético representado pela Figura 5.2, em que os processos p_e e p_f executam a função $Leader()$. As operações de escrita representadas referem-se a escritas executadas por p_a, p_b, p_c e p_d nos registradores $Punishments[-][x]$ e $Punishments[-][y]$ na linha 41 (e.g., w_x^a denota a operação $write(Punishments[a][x], -)$). Também são exibidas as operações de leitura realizadas por p_e e p_f sobre os mesmos registradores, relativas à linha 45 (e.g., r_x^d denota a operação $read(Punishments[d][x])$. Note que, uma vez que operações de leitura se sobreponham a operações de escrita, é possível que algumas das leituras invocadas por p_e e p_f obtenham os valores que estão sendo escritos concorrentemente, enquanto outras não. Assim, é possível que $Leader()$ retorne x para p_e no instante t'' e y para p_f no mesmo instante.

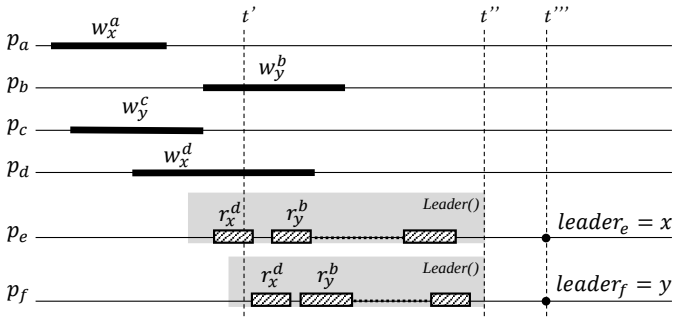

Figura 5.2 p_e e p_f obtém visões distintas do sistema, enquanto executam a função $Leader()$

Considere, então, que neste cenário, no instante t''', menos de α p_j consideram p_x como líder. Se nenhuma nova escrita é invocada após t', em algum momento após a última escrita ser concluída, todos os processos que não falham veem o mesmo estado S_k de *Punishments* de modo que, ao invocar *Leader*() (depois da última escrita) em Task 1 (linha 5), recebem k. Em Task 2, ao completar uma iteração do **for** (linhas 30-35) sem ter incluído pelo menos α processos em $updated_i$, p_i atribui k à variável local ld_i (linhas 36-37). Se p_k é estável, ao cabo de um tempo, p_i obtém $Alive[k]$ atualizado (linha 25 ou 32) e a *Condição 1* é satisfeita. Caso contrário, todo $p_h \in STABLE$ suspeita de p_k e atualiza $Alive[h]$ (linhas 14-16 em Task 1), de modo que pelo menos α $Alive[h]$ são encontrados atualizados e a *Condição 2* é satisfeita.

Para que novas punições ocorram, também é necessário que pelo menos α processos p_h suspeitem de p_k e atualizem $Alive[h]$ em Task 1 (linhas 12-16). Da mesma forma, p_i acaba obtendo α $Alive[h]$ atualizados, inclui cada p_h em $updated_i$ e a *Condição 2* é satisfeita.

Caso 4: p_x é estável, mas não se considera líder.

Logo, p_x não atualiza $Alive[x]$ na linha 8, mas pode fazer isso na linha 16, se não obtiver o registrador $Alive$ de seu líder atualizado. Tem-se então:

Caso 4.1: p_x atualiza $Alive[x]$. Portanto, depois de um tempo, p_i obtém o valor atualizado e a *Condição 1* é satisfeita.

Caso 4.2: p_x não atualiza $Alive[x]$. Então, como no Caso 2, se pelo menos α processos p_j consideram p_x como líder, a *Condição 2* é satisfeita. Caso contrário, tem-se os vários desdobramentos do Caso 3, e depois de um tempo, uma das duas condições é satisfeita. $Lema 5.3 \square$

Lema 5.4. *Se um processo p_x é considerado líder por algum processo p_i e falha no instante t, então, existe um instante $t' > t$, em que p_x é demovido da posição de líder, e a partir de um instante $t'' \geq t'$, p_x não será considerado líder por qualquer processo.*

Prova.

Considere os seguintes casos:

Caso 1: Pelo menos α processos p_j consideram p_x como líder em Task 1 (linha 5) quando p_x falha.

Como na prova do Caso 2 do Lema 5.3, em algum instante, p_i deixa o laço **while** com $| updated_i | \geq \alpha$ e $p_x \notin updated_i$ (linha 24). Então p_i avalia o predicado da linha 38 como verdadeiro e pune p_x, deixando pelo menos α p_j impunes. Na próxima vez que p_i invoca *Leader*(), se $soma_i[x]$ não tiver ultrapassado $soma_i[j]$ para algum j (ou igualado para algum $j < x$), e pelo menos α p_j ainda considerarem p_x como líder, o procedimento acima se repete até que $MinLex(soma_i[j]) \neq x$ (linha 47). Assim, p_x é demovido da posição de líder de p_i.

Se, no entanto, menos de α p_j, incluindo p_i, permanecem confiando em p_x, tem-se o seguinte caso.

Caso 2: Menos de α processos consideram p_x como líder em Task 1 (linha 5), quando p_x falha.

Mais uma vez recorre-se à prova do Lema 5.3; desta vez, para o Caso 3. Como existem líderes concomitantes a p_x (pelo menos um), sabe-se que há operações de escrita em *Punishments* concluídas dentro de um intervalo $I = [t_1, t_2]$, que não foram vistas por todos os processos. Portanto, a invocação de *Leader*() pelo mesmo processo em vários instantes de I pode retornar líderes distintos. Tem-se então os seguintes casos:

Caso 2.1: p_i define um novo líder em algum instante $t' \in I$, ou seja, p_x é demovido da posição de líder.

Caso 2.2: Em algum instante $t'' \in I$, pelo menos α p_j, incluindo p_i, mais uma vez, definem p_x como líder. Este é o Caso 1, em que p_x é demovido da posição de líder.

Portanto, se um processo p_x, que é considerado como líder por p_i, falha no instante t, será demovido dessa posição no instante $t' > t$. Além disso, desde o instante em que p_x falha, ele não atualiza mais $Alive[x]$, e então p_x deixará de ser incluído em qualquer $updated_j$. Assim, sempre que algum p_j suspeitar de seu líder e o punir, p_x será punido também – a mesma quantidade de penalidades impostas a p_j será imposta a p_x.

Suponha que após t', p_i define p_y como seu líder (porque $soma_i[y] < soma_i[x]$ ou $soma_i[y] = soma_i[x]$ e $y < x$). Então, se p_y é estável e não é punido em qualquer instante, permanece líder e o Lema se mantém. Se um p_j, pune p_y, p_x também será punido (linhas 38-41) de modo que a relação acima entre os montantes das punições fica inalterada. Então, a partir de um instante $t'' \geq t'$, p_x não será considerado líder por qualquer processo. $_{Lema5.4}\square$

Teorema 5.1. *Existe um instante após o qual algum $p_l \in STABLE$ será eleito permanentemente líder.*

Prova.

Segue direto dos Lemas 5.2, 5.1, 5.3 e 5.4. $_{Teorema5.1}\square$

5.5 COMPLEXIDADE

O algoritmo usa n registradores *Alive* e n^2 registradores *Punishments*, isto é, $O(n^2)$ registradores $1WMR$ em cada execução, onde n é o número de processos que se liga ao sistema. Uma vez que o número de rodadas não é limitado, o tamanho dos registradores *Alive* é ilimitado também. No entanto, os registradores *Punishments* são limitados, já que a partir do instante em que PAM é satisfeita, nenhum processo mais é punido.

Com relação à complexidade de passo, tem-se que, inicialmente, um processo p_i invoca a função *Leader*() para definir um líder de acordo com o nível de suspeições corrente. Ao executar *Leader*(), são realizadas $[n(t)]^2$ operações de leitura sobre *Punishments*, onde $n(t)$ é o número de processos que já se ligaram à computação distribuída até o instante t. Nenhuma escrita é executada durante a execução de *Leader*().

Em TASK 0, p_i executa 1 operação de escrita cada vez que um novo processo p_r entra no sistema. Nenhuma leitura é feita nesta tarefa.

Em TASK 1, p_i executa $[n(t)]^2$ operações de leitura (em $Leader()$, invocada na linha 5) e então:

1. Se p_i considera-se líder ($leader_i = i$), executa 1 escrita em $Alive[i]$.

2. Se p_i considera p_j como líder ($leader_i = j$), executa 1 leitura em $Alive[j]$ e pode ter que executar 1 escrita (linhas 12-16). Isto é, p_i executa, no máximo, 1 escrita.

Em TASK 2, p_i executa $[n(t)]^2$ operações de leitura (em $Leader()$, invocada na linha 21) e então:

1. Se p_i considera-se líder, não executa qualquer acesso à memória compartilhada (linhas 21-22).

2. Se p_i considera p_j como líder ($leader_i = j$), executa:

   (a) 1 leitura em $Alive[j]$ e nenhuma escrita, caso obtenha $Alive[j]$ atualizado nesta leitura (linhas 25-28); ou

   (b) pelo menos $n(t)$ leituras sobre $Alive[-]$, se obtém $Alive[j]$ atualizado em alguma dessas leituras (linhas 30-35) e nenhuma escrita; ou

   (c) no pior caso, p_i suspeita de p_j e então executa, pelo menos $n(t)$ leituras sobre $Alive[-]$ (linhas 30-35) e tantas punições quantos forem os registradores encontrados desatualizados, isto é, até $n(t) - \alpha$ escritas em $Punishments$ (linhas 38-41).

Portanto, a partir do instante em que a propriedade 5.2.3 é satisfeita, e considerando que a chegada de processos cessou, em uma rodada, um processo p_i executa, no pior caso, $O([n(t)]^2)$ leituras e 1 escrita.

Focando num sistema onde leituras são locais (BALDONI et al., 2009) (ou mesmo mais baratas que escritas), o custo real do algoritmo deve-se às operações de escrita. Por esse motivo, a abordagem utilizada no projeto do algoritmo foi a redução das escritas, o que de fato foi obtido a partir da primeira versão do algoritmo.

Considerando execuções *bem-comportadas*, nas quais o registrador do líder é sempre encontrado atualizado, apenas o líder eleito permanece escrevendo para sempre na memória compartilhada. O mesmo ocorre no caso especial em que $\alpha = 1$. Nessas situações, o comportamento do algoritmo é ótimo, já que em qualquer algoritmo de Ω para o modelo proposto, pelo menos um processo, o líder eleito, p_l, precisa informar seu progresso pra sempre. Caso contrário, não há como detectar uma possível falha de p_l, o que pode levar os processos no sistema a confiarem em um líder faltoso (FERNÁNDEZ et al., 2010; GUERRAOUI; RAYNAL, 2006).

5.6 TRABALHOS RELACIONADOS

A Tabela 5.1 resume as principais abordagens para eleição eventual de líder e implementação de Ω em sistemas assíncronos propostas na literatura. Pode-se distinguir duas grandes categorias: (i) as que estendem o sistema assíncrono com requisitos temporais (a grande maioria) e são consideradas "baseadas em temporizadores" (*timer-based*); e (ii) as que não usam o tempo, mas um padrão de troca de mensagens ou de acesso à memória, "baseadas em padrão de mensagens" ou "livres de tempo", (*message-pattern* ou *time-free*). Como a Tabela 5.1 mostra, a maioria considera o modelo de passagem de mensagens e poucos são para o modelo de memória compartilhada.

Abordagem	Comunicação	Π	Conectividade	Propriedade do Modelo
(CHANDRA; HADZILACOS; TOUEG, 1996)	passagem de mensagens	conhecido	grafo completo	sistema parcialmente síncrono
$\exists p$ *eventually accessible* (AGUILERA et al., 2001)	passagem de mensagens	conhecido	grafo completo	$\exists p$:todos os canais bidirecionais de/para p são *eventually timely*
$\exists p \diamondsuit f$-*source* (AGUILERA et al., 2004)	passagem de mensagens	conhecido	grafo completo	$\exists p$: f canais de saída de p $\exists p$: f são *eventually timely*
$\exists p \diamondsuit f$-*accessible* (MALKHI; OPREA; ZHOU, 2005)	passagem de mensagens	conhecido conhecido	grafo completo	$\exists p, Q$: *round-trip* de p para $\forall q \in Q$, é *eventually timely*
$\exists p$ *eventually moving* f-*source* (HUTLE et al., 2009)	passagem de mensagens	conhecido	grafo completo	$\exists p, Q$: canal de saída de p para $\forall q \in Q$, é *eventually timely*
(JIMÉNEZ; ARÉVALO; FERNÁNDEZ, 2006)	passagem de mensagens	desconhecido dinâmico	nós corretos conectados	$\exists p$ correto: $\forall q$ correto, pode ser atingido de p por caminho *eventually timely*
(ANTA; JIMÉNEZ; RAYNAL, 2010)	passagem de mensagens	desconhecido dinâmico	nós corretos conectados	$\exists p$ correto e mais $(n$-$f)$-α processos corretos alcançáveis de p por canais *eventually timely*
(GUERRAOUI; RAYNAL, 2006)	memória compartilhada	conhecido	–	sistema síncrono
well behaved timer (FERNÁNDEZ et al., 2010)	memória compartilhada	conhecido	–	$\exists p$ cujos acessos à memória compartilhada são *eventually timely*
mecanismo *query-response* (*time-free*) (MOSTEFAOUI; MOURGAYA; RAYNAL, 2003)	passagem de mensagens	conhecido	grafo completo	$\exists t, p, Q$: a partir de t, $q \in Q$: recebe uma resposta de p à suas *queries* entre as $(n-f)$primeiras.
mecanismo *query-response* (*time-free*) (ARANTES et al., 2013)	passagem de mensagens	desconhecido dinâmico	nós estáveis conectados	$\exists t, p, Q$: a partir de t, $q \in Q$ recebe uma resposta de p à suas *queries* entre as α primeiras.
$\Delta^*\Omega$ (GOMEZ-CALZADO et al., 2013)	passagem de mensagens	desconhecido dinâmico	grafo dinâmico	Em "bons" períodos, $\exists$ um limite δ sobre o tempo de transmissão em todos os links.
PAM-padrão de acesso à memória (KHOURI; GREVE, 2014)	memória compartilhada	desconhecido dinâmico	–	$\exists t, p_l$: a partir de t, $Alive[l]$ está entre os α primeiros registradores atualizados

Tabela 5.1 Abordagens para Implementação de Ω

5.6.1 Soluções Clássicas para Modelo de Sistemas Estáticos

A busca por requisitos mínimos relativos ao grau de sincronia e de confiabilidade dos canais e, mais recentemente, seu grau de conectividade, foram fruto de estudo de diversos

trabalhos para o modelo de passagem de mensagem. As 7 entradas iniciais da Tabela 5.1 adotam tal abordagem.

Os primeiros estudos de detectores de líder consideravam uma rede de comunicação completamente conectada com todos os canais confiáveis e síncronos a partir de um determinado instante (CHANDRA; HADZILACOS; TOUEG, 1996). Já (AGUILERA et al., 2001) considera que existe pelo menos um processo correto p e um instante t, tais que a partir de t, todos os canais bidirecionais entre p e todos os outros processos são síncronos (*eventually timely*). Em (AGUILERA et al., 2004), os autores mostram que Ω pode ser implementado em sistemas com falhas por parada, cujos canais sejam *fair-lossy* (permitem perdas equitáveis), desde que pelo menos um processo correto p, possua f canais de saída que sejam síncronos após um tempo (*eventually timely*). Nesse caso, p é dito ser um $\diamondsuit f$-*source*.

Em (MALKHI; OPREA; ZHOU, 2005), a estratégia adotada é diferente das anteriores; eles usam a noção de *eventually* f-*acessibility*. Supõe-se que existe um instante t, um processo p e um conjunto Q de f processos, tais que, $\forall t' \geq t$, p troca mensagens com cada $q \in Q$ dentro do limite de tempo δ (relativo ao *round-trip* do canal), isto é, p é $\diamondsuit f$-*accessible*. Deve-se observar que o subconjunto de processos Q não é fixo, isto é, pode variar com o tempo. Uma suposição ainda mais fraca é a proposta em (HUTLE et al., 2009): existe um processo correto p, a saber, um *eventually moving f-source*, tal que, após um tempo, cada mensagem que ele envia é recebida em tempo (*timely*) por um conjunto Q de f processos que pode ser diferente em instantes distintos.

No que diz respeito ao paradigma de comunicação por memória compartilhada, poucas são as propostas encontradas para Ω e a maioria delas considera sistemas estáticos. (GUERRAOUI; RAYNAL, 2006) implementam um protocolo com registradores regulares compartilhados, mas diferentemente deste trabalho, consideram um sistema estático, síncrono após um tempo. (FERNÁNDEZ; JIMÉNEZ; RAYNAL, 2007) e (FERNÁNDEZ et al., 2010), consideram um sistema assíncrono com registradores atômicos compartilhados; mas, ao contrário da presente proposta, supõem um ambiente estático, além de canais síncronos.

5.6.2 Soluções Recentes para Modelo de Sistemas Dinâmicos

Os trabalhos anteriormente citados consideram um modelo clássico de sistemas distribuídos, cujos participantes são conhecidos. Jiménez et al (JIMÉNEZ; ARÉVALO; FERNÁNDEZ, 2006) propõem um protocolo para ambiente dinâmico com troca de mensagens, supondo canais síncronos após um tempo. Fernández et al (ANTA; JIMÉNEZ; RAYNAL, 2010) apresentam um protocolo considerando que os processos não conhecem o número de participantes n nem o limite para o número de falhas f. Como na abordagem aqui considerada, cada processo só conhece sua própria identidade e um limite inferior α sobre o número de processos corretos.

Gomez-Calzado et al (GOMEZ-CALZADO et al., 2013) propõem um modelo básico para sistemas distribuídos dinâmicos, que leva em conta a mobilidade dos nós, a partir de sete parâmetros, e consideram uma configuração particular para abordar o problema da eleição de líder após um tempo. Nesta configuração, eles assumem que todos os

parâmetros, entre os quais, o número de processos participantes, alternam períodos entre limitados e ilimitados. Uma redefinição do problema da eleição de líder após um tempo também é apresentada, apropriada a este modelo de sistema. Como resultado, eles apresentam a classe de detectores *Ômega Dinâmico Móvel*, ou $\Delta^*\Omega$, e um algoritmo que implementa um detector da referida classe.

Diferentemente da proposta aqui apresentada, entretanto, as abordagens acima assumem que os processos se comunicam por troca de mensagens e o algoritmo, baseado em temporizadores, exige a existência de alguns canais síncronos após um tempo ou, pelo menos, em períodos alternados.

Em um sistema de memória compartilhada cujos participantes são desconhecidos, um processo que entra na computação, naturalmente, precisa inteirar-se do estado do sistema antes de participar efetivamente. (BALDONI et al., 2009) propõem uma abstração de registrador regular compartilhado no topo de um sistema dinâmico de passagem de mensagens. Quando um processo p_i deseja entrar, invoca uma operação $join()$, que lhe permite obter um estado consistente do registrador. De modo semelhante, o protocolo ora proposto envolve a invocação de uma operação $join()$ que se encarrega de criar registradores devidamente inicializados.

Abordagens Livres de Tempo Baseadas em Padrão de Mensagens.

A fim de possibilitar a implementação de detectores num sistema assíncrono, (MOSTEFAOUI; MOURGAYA; RAYNAL, 2003; MOSTEFAOUI et al., 2006) introduziram uma nova abordagem baseada num mecanismo pergunta-resposta (*query-response*), o qual assume que a resposta de algum p_i às últimas consultas requisitadas por p_j está sempre entre as primeiras $n - f$ respostas recebidas por cada p_j. A intuição por trás da suposição é a de que, mesmo que o sistema não exiba propriedades síncronas, ele pode apresentar alguma regularidade de comportamento que se traduza numa *sincronia lógica* capaz de contornar a impossibilidade de conversão do algoritmo.

Considerando uma suposição semelhante, (ARANTES et al., 2013) propõem um modelo para implementação de Ω num sistema dinâmico de passagem de mensagens que considera a mobilidade dos nós. Supondo um padrão de mensagens adaptado a sistemas dinâmicos, segundo o qual os processos envolvidos pertencem a um conjunto de nós *estáveis*, eles utilizam o limite inferior α sobre o número de processos estáveis para controlar o número de respostas consideradas entre as primeiras. Essa constante α abstrai todos os pares (n, f) tradicionalmente considerados como conhecidos, tal que $n - f \geq \alpha$. Dessa forma, um protocolo baseado em α, funciona para qualquer par (n, f).

Mais recentemente, (KHOURI; GREVE, 2014) apresentam um protocolo para implementar Ω num modelo de memória compartilhada que, semelhantemente ao protocolo aqui apresentado, tem por base um padrão de acesso à memória compartilhada. Ao que se sabe, não existe uma implementação de Ω para memória compartilhada baseada em suposições livre de tempo. Nesse sentido, o trabalho é pioneiro.

Entretanto, no protocolo em (KHOURI; GREVE, 2014), todos os processos continuam a escrever nos registradores para sempre, mesmo após a eleição do líder. O protocolo aqui apresentado avança na obtenção de uma solução ótima para as escritas, reunindo condições para que somente o líder, após um tempo, continue a escrever no seu registrador.

Com tal estratégia, semelhantemente às soluções em (AGUILERA et al., 2004; MALKHI; OPREA; ZHOU, 2005; HUTLE et al., 2009) para passagem de mensagens, que visavam optimalidade na quantidade de canais síncronos e de processos emissores de mensagens, avançou-se no estado da arte da implementação de Ω para memória compartilhada. Em ambos os casos, o objetivo de fazer com que somente o líder atue (emita mensagens ou escreva no registrador, após um tempo) é atingido.

5.7 CONSIDERAÇÕES FINAIS

Este trabalho avança na compreensão do problema de eleição de líder em memória compartilhada, considerando um sistema dinâmico sem requisitos temporais explícitos. Em resumo, as principais contribuições deste capítulo são:

1. A definição de um modelo de sistema dinâmico sujeito a uma propriedade livre de tempo, baseada num padrão de acesso à memória, que permite a implementação de um serviço de líder numa memória compartilhada assíncrona, onde o conjunto de participantes é desconhecido;

2. Um protocolo que implementa um serviço de líder da classe Ω adequado ao modelo proposto e que visa optimalidade na escrita dos registradores. O sistema dinâmico considera a existência de um conjunto de processos estáveis (entram no sistema e permanecem ativos sem falhar ou sair), cuja cardinalidade mínima é α. O algoritmo proposto tem resiliência ótima, já que mantém a corretude se ao menos um único processo estável permanece no sistema ($\alpha \geq 1$).

Muitas questões permanecem abertas nessa área. Dando continuidade a este estudo, pretende-se trabalhar na implementação de um sistema de memória compartilhada sobre um sistema assíncrono dinâmico. Nesse sentido, busca-se refletir em questões como: Que suposições considerar para estender o sistema assíncrono de modo a permitir a implementação de registradores (atômicos/regulares)? É possível tal implementação a partir de um modelo de padrão de troca de mensagens, de modo a dispensar o uso de temporizadores nos protocolos? Quais os requisitos para a implementação de uma operação $join()$?

Quanto à implementação específica do detector Ω, questiona-se: que tipos de objetos compartilhados seriam mais adequados para a implementação de um detector Ω num ambiente assíncrono dinâmico? Que requisitos de consistência devem ser considerados? Quais os limites de complexidade de espaço para esse problema? Quais as suposições mais fracas sobre a estabilidade do sistema que permitem a resolução do problema? São questões que ainda estão por serem respondidas e que, pela sua importância, devem ser investigadas.

Capítulo

6

São apresentadas aqui algumas considerações sobre o trabalho realizado, destacando-se suas principais contribuições e apontando alguns caminhos para futuros trabalhos.

CONSIDERAÇÕES FINAIS

Nesta tese foi estudado o problema do consenso em um sistema assíncrono com participantes desconhecidos, considerando o modelo de memória compartilhada para a comunicação entre processos. Como resultado do estudo, foi apresentado um conjunto de protocolos para a resolução do consenso em tal ambiente. Para lidar com o dinamismo do sistema foram consideradas duas abordagens:

(1) uso de detectores de participantes; e

(2) uso de um arcabouço de memória compartilhada.

Um detector de participantes provê os processos com um conhecimento inicial sobre os participantes da computação. Tal conhecimento pode ser modelado como um grafo da conectividade do conhecimento cujas propriedades definem a classe do detector de participantes. Foi apresentado um conjunto de algoritmos com as respectivas provas de que eles funcionam corretamente para sistemas equipados com um detector de participantes da classe k-OSR, um detector de líder da classe Ω e registradores regulares do tipo $1WMR$ (um escritor e múltiplos leitores), admitindo até $f < k < n$ falhas por parada. A classe k-OSR é a que fornece as condições minimais para a resolução do consenso tolerante a falhas com participantes desconhecidos – FT-CUP – em um sistema assíncrono estendido com um detector de falhas ou de líder. Por outro lado, as classes de detectores de falhas, $\diamondsuit\mathcal{S}$, e de líder, Ω, são as que fornecem o sincronismo mínimo necessário para possibilitar a resolução do consenso nesse ambiente. Portanto, com relação às condições de conectividade do conhecimento e de sincronismo, o protocolo proposto é ótimo.

A segunda abordagem consiste em abstrair o gerenciamento da dinamicidade, delegando esta responsabilidade à camada de implementação da memória compartilhada. No caso, o sistema de memória compartilhada se incumbe de fornecer a um processo recém-chegado, um estado consistente da memória. O projeto dos algoritmos, por sua vez, considera dois requisitos básicos: o fato de que o valor de n não está disponível, de

modo que o algoritmo tem que ser independente de n; e a existência de um limite inferior para o número de processos *estáveis*. Os algoritmos assim concebidos funcionam corretamente considerando as duas abordagens. Foi apresentado então um outro protocolo de consenso, o qual é genérico com relação ao detector de falhas que utiliza, e que não depende do conhecimento prévio do número de participantes.

Ambos os protocolos necessitam de um detector de líder que funcione em um ambiente dinâmico. Assim, foi proposto um protocolo que implementa um detector de líder da classe Ω baseado no mesmo arcabouço de memória compartilhada para sistemas assíncronos dinâmicos. Conforme as provas apresentadas, o algoritmo funciona corretamente sem que os processos precisem conhecer *a priori* o número de participantes.

Os resultados alcançados contribuem para a concepção de soluções práticas com foco em sistemas tolerantes a falhas onde os processos compartilham algum meio de armazenamento e usam o consenso ou um serviço de eleição de líder como blocos de construção. Por exemplo, sistemas multinúcleo, onde processadores (núcleos) compartilham uma memória física comum além de sua evolução em direção a sistemas *exascale*. Nessas arquiteturas, são executados programas multiprocessos compostos por processos assíncronos sujeitos a falhas (HERLIHY; LUCHANGCO, 2008; RAYNAL; STAINER, 2012).

Outra aplicação interessante diz respeito aos sistemas de armazenamento de dados como as redes de área de armazenamento (SANs) em que vários conjuntos de discos são conectados em rede (NADs) e podem ser acessados diretamente pelos processos. Nesses sistemas distribuídos, processos se comunicam através de operações de escrita e leitura sobre discos, os quais implementam uma memória compartilhada.

6.1 CONTRIBUIÇÕES

No decorrer do estudo, foram feitas algumas contribuições pontuais materializadas em artigos publicados nos últimos anos que culminaram com o documento aqui apresentado. As contribuições resultantes coincidem com o cumprimento dos objetivos previamente definidos.

6.1.1 Definição de um modelo adequado para abordar o problema do consenso em ambientes dinâmicos

O modelo considerado foi o de um sistema assíncrono de memória compartilhada em que o conjunto de participantes é desconhecido. O número de processos no sistema é infinito, no entanto, em cada execução, admite-se um número limitado, n, de participantes, significando que a partir de dado momento, os processos cessam de chegar (este é o *modelo de chegada finita* (AGUILERA, 2004)).

Para o primeiro conjunto de protocolos produzido, que resolve o SM FT-CUP, o número de faltas tolerado é tal que $f < k < n$, onde k é definido pelo detector de participantes da classe k-OSR e, consequentemente, pelo *grafo da conectividade do conhecimento* induzido por ele.

Para a implementação do consenso genérico e de Ω, o número de participantes no sistema, n, e o limite para o número de falhas, f, são abstraídos por um parâmetro α,

utilizado para modelar a dinamicidade, o qual representa um limite inferior para o número de processos estáveis no sistema. O algoritmo genérico independe deste parâmetro, no sentido de que os processos não precisam saber o valor de α para garantir a terminação do protocolo. Por outro lado, no algoritmo de Ω, o valor de α é utilizado pelos processos como uma condição de progresso e como tal, deve ser conhecido dos participantes.

Em ambos os casos, no entanto, a condição $\alpha \geq 1$ significa que pelo menos um processo precisa permanecer ativo para sempre a fim de garantir a terminação e a estabilidade da eleição. Essa condição também indica que os algoritmos produzidos para o modelo são *wait-free*. Deve-se observar, entretanto, que existe um conflito entre a escolha do valor de α e o desempenho do algoritmo de Ω.

Para viabilizar a implementação de Ω, foi proposta uma propriedade que define um padrão de acesso à memória – PAM, a qual, juntamente com a utilização de α, contorna a impossibilidade causada pelo assincronismo do sistema. A propriedade PAM é *time-free*, de modo que o algoritmo desenhado para o modelo de sistema munido com essa propriedade não depende de temporizadores.

Neste trabalho foram exploradas as classes de detectores de falhas *forte após um tempo* – $\diamondsuit\mathcal{S}$ (*eventually strong*) (CHANDRA; TOUEG, 1996) e *líder após um tempo* (*eventual leader*) (CHANDRA; HADZILACOS; TOUEG, 1996). Num ambiente dinâmico, a definição desses oráculos deve levar em conta não somente as falhas mas também a saída arbitrária de processos.

Um detector $\diamondsuit\mathcal{S}$, quando invocado por um processo, fornece uma lista de processos suspeitos. Num contexto dinâmico, $\diamondsuit\mathcal{S}$ deve satisfazer às seguintes propriedades:

Completude forte (*Strong completeness*): Após um tempo, todo processo faltoso ou que abandona o sistema 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.

Um detector Ω, quando requisitado por um processo, retorna a identidade de um processo considerado correto. Num contexto dinâmico, Ω satisfaz à seguinte propriedade:

Liderança após um tempo (*Eventual leadership*): existe um instante após o qual qualquer invocação de $Leader()$ por qualquer processo $p_i \in STABLE$ retorna a mesma identidade de processo $p_l \in STABLE$.

6.1.2 Conjunto de algoritmos para resolução do consenso tolerante a falhas com participantes desconhecidos em um sistema de memória compartilhada – SM FT-CUP

A maioria dos protocolos encontrados na literatura para o problema do consenso destina-se a sistemas em que o conjunto de participantes é conhecido e os processos se comunicam trocando mensagens. Alguns trabalhos propõem solução para o consenso em ambientes

dinâmicos com passagem de mensagens, outros consideram sistemas de memória compartilhada com participantes conhecidos. Não foram encontrados algoritmos para a resolução do consenso tolerante a falhas com participantes desconhecidos em que a comunicação se dá através de memória compartilhada – SM FT-CUP. Assim, foi proposto um protocolo para atender a esta demanda.

Diferentemente dos sistemas clássicos, onde o conjunto de participantes e suas identidades são conhecidos, em ambientes dinâmicos não se sabe *a priori* quais são os pares com quem se pode colaborar, nem quantos deles estão disponíveis. Um primeiro esforço no sentido de atacar o problema foi a elaboração de um conjunto de protocolos que com a ajuda de um detector de participantes (PD) da classe k-OSR, habilita os processos a atingirem o consenso. Esse primeiro resultado foi publicado em (KHOURI; GREVE; TIXEUIL, 2013).

A informação fornecida por um PD pode ser modelada como um grafo da conectividade do conhecimento – G_{di}. No caso do PD k-OSR, o grafo acíclico direcionado (DAG), obtido pela redução de G_{di} a seus componentes fortemente conectados, contém exatamente um *poço*, denotado G_{sink}. Foi apresentado então um conjunto de protocolos que permite a cada processo p_i:

1. Ampliar sua visão do sistema a partir do conhecimento inicial provido pelo PD (COLLECT);

2. Definir sua posição no grafo da conectividade do conhecimento, isto é, p_i descobre se pertence ao componente poço ou não (SINK);

3. Executar um algoritmo de consenso, caso $p_i \in G_{sink}$; ou obter a decisão dos componentes do poço, caso contrário (CONSENSUS).

Uma vez que os elementos do poço são conhecidos entre si, eles podem executar um algoritmo de consenso clássico baseado em um oráculo detector de falhas ou de líder. 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. Tal implementação do consenso pode ser melhor adaptada às características particulares de cada ambiente; principalmente quando a implementação do detector serve a muitas aplicações. Assim, as aplicações existentes que já estão rodando sobre os detectores específicos podem ser portadas mais facilmente.

Não foi identificado na literatura um protocolo de consenso que atendesse a esses requisitos. Desse modo, foi desenvolvido um algoritmo de consenso *wait-free*, genérico quanto ao oráculo, na medida em que pode ser instanciado com um detector de falhas $\diamondsuit\mathcal{S}$ ou Ω. O algoritmo produzido, GENERICON, considera um conjunto de participantes, Π, fixo e conhecido, isto é, conta com o conhecimento da cardinalidade de Π, n. Duas versões do algoritmo foram produzidas (KHOURI; GREVE, 2013a, 2013b) e até onde se sabe, é o primeiro algoritmo de consenso para memória compartilhada, genérico quanto ao oráculo.

6.1.3 Algoritmo genérico para a solução do consenso, sem conhecimento prévio de n

O algoritmo de consenso desenhado para o SM FT-CUP pressupõe o conhecimento de n, o que num contexto real de sistema dinâmico, nem sempre é possível. Logo, é importante a disponibilidade de algoritmos que funcionem corretamente independentemente desse conhecimento. Assim como no caso anterior, não foram identificados algoritmos genéricos quanto ao oráculo destinados a realizar o consenso em sistemas de memória compartilhada em que n não está disponível. Dessa maneira, parte deste trabalho dedicou-se ao desenvolvimento de tal algoritmo.

Diferente da proposta anterior, o algoritmo então produzido não pressupõe conhecimento de n, podendo ser usado de maneira plena e direta no ambiente dinâmico. Ao que se sabe, este é o primeiro protocolo de consenso genérico quanto ao oráculo para sistemas de memória compartilhada que não requer o conhecimento prévio de n. O algoritmo é *wait-free*, o que significa que é ótimo com relação à resiliência.

Na revisão da literatura, não foi identificado um limite para a complexidade de espaço de protocolos de consenso no ambiente aqui considerado. Sabe-se que num ambiente estático de memória compartilhada o limite inferior para o número de registradores atômicos é n (LO; HADZILACOS, 1994). O algoritmo proposto neste trabalho usa n registradores, um para cada processo que se liga ao sistema. Uma vez que o número de rodadas não é limitado, entretanto, o tamanho dos registradores é ilimitado também.

Quanto à complexidade de passo, quando um processo p_i é o proponente da rodada, executa, no pior caso (quando p_i atinge uma decisão), duas leituras sobre o arranjo de registradores e três escritas sobre $R[i]$, na rodada. Um processo p_j que não é proponente mantém-se lendo o registrador do proponente até que este seja considerado suspeito ou alguma decisão seja obtida. Se é obtida uma decisão válida, apenas uma escrita é realizada por p_j que encerra a execução. Caso contrário, o custo para o início de uma nova rodada varia com o oráculo. Se $\mathcal{D} \in \Omega$, p_j inicia a nova rodada sem fazer qualquer acesso à memória compartilhada. Se, no entanto, $\mathcal{D} \in \diamondsuit\mathcal{S}$, p_j terá que fazer uma leitura sobre o arranjo de registradores.

Em execuções bem-comportadas, para atingir uma decisão, o proponente p_i executa três operações de escrita e duas operações de leitura sobre o arranjo de registradores enquanto que os demais processos mantêm-se lendo apenas o registrador do proponente até obter a decisão, quando então a escrevem em seu registrador.

6.1.4 Algoritmo para implementação de Ω independente do conhecimento prévio de n

O detector Ω é um mecanismo fundamental utilizado em vários serviços de acordo para sistemas tolerantes a falhas. O algoritmo de consenso genérico apresentado nesta tese, por exemplo, pode ser instanciado com um detector Ω. Dada a sua importância, e em benefício da completude da solução aqui apresentada para o consenso, foi apresentado um algoritmo para a implementação deste oráculo no ambiente dinâmico considerado.

Foram encontradas algumas implementações de Ω na literatura, mas assim como para o consenso, a maioria delas é dedicada a sistemas estáticos de passagem de mensagem.

Além disso, a totalidade das propostas voltadas para sistemas de memória compartilhada considera suposições temporais para estender o modelo de sistema assíncrono de modo a viabilizar a terminação dos protocolos. Não foi encontrada qualquer abordagem que considere suposições livres de tempo. Assim, foi apresentado um algoritmo para implementação de Ω em um ambiente dinâmico de memória compartilhada assíncrono enriquecido com uma propriedade comportamental relativa aos acessos à memória compartilhada. O protocolo assim produzido é dito *livre de tempo* (*time-free*), uma vez que não utiliza temporizadores para garantir progresso.

A propriedade assumida considera um padrão de acesso à memória semelhante a uma proposta relativa a um padrão de mensagem introduzida em (MOSTEFAOUI; MOURGAYA; RAYNAL, 2003) e baseia-se em duas suposições:

1. Existe um parâmetro α que corresponde ao número mínimo de processos *estáveis* no sistema, tal que $\alpha \geq 1$.

2. Existe um instante t e um processo estável p_l tais que, a partir de t, o registrador $Alive[l]$ vai estar sempre entre os α primeiros encontrados atualizados por todo processo no sistema.

O algoritmo proposto funciona corretamente desde que as suposições acima sejam satisfeitas. Numa primeira versão, publicada em (KHOURI; GREVE, 2014), todos os processos escrevem permanentemente na memória compartilhada. Após revisão, o algoritmo foi otimizado de modo que num período instável, no pior caso, um processo p_i executa até $n(t) - \alpha$ escritas, onde $n(t)$ é o número de processos que entrou no sistema até o instante t; e $[n(t)]^2$ leituras. A partir do instante em que a propriedade PAM é satisfeita, nenhum processo mais será punido, de modo que cada p_i executa, em uma rodada, $[n(t)]^2$ leituras e até 1 escrita. Numa execução *bem comportada*, em que todo p_i sempre encontra $Alive[leader_i]$ atualizado, apenas p_l permanece escrevendo pra sempre na memória compartilhada.

No total, o algoritmo usa n registradores $Alive$ e n^2 registradores $Punishments$, isto é, $O(n^2)$ registradores $1WMR$, onde n é o número de processos que se liga ao sistema. Os registradores $Alive$ são ilimitados, porém os registradores $Punishments$ são limitados, já que a partir do instante em que PAM é satisfeita, nenhum processo mais é punido.

6.2 TRABALHOS FUTUROS

O estudo realizado levantou novas questões de interesse nas quais espera-se trabalhar num futuro próximo.

6.2.1 Implementação da memória compartilhada dinâmica

Para o desenvolvimento dos algoritmos de líder e de consenso com participantes desconhecidos (independente de n), foi considerado que a memória dinâmica é implementada sobre uma camada de sistema em que os processos se comunicam através de troca de mensagens. Assumiu-se que a camada inferior provê uma primitiva $join()$ que fornece a

um processo recém-chegado um estado consistente da memória, isto é, dos registradores compartilhados.

O trabalho apresentado em (BALDONI; BONOMI; RAYNAL, 2012) propõe a implementação de um registrador regular $MRMW$ em um sistema dinâmico síncrono após um tempo (*eventually synchronous*). Eles fornecem as operações primitivas $join()$, $read()$ e $write()$. Em um artigo anterior, os mesmos autores consideram o registrador regular $MWMR$ e o objeto conjunto (*set*) em um mesmo ambiente e fornecem uma implementação para a operação $join()$. Em ambos os trabalhos é considerado um modelo de *churn* introduzido em (BALDONI; BONOMI; RAYNAL, 2010a).

Não foi encontrada na literatura uma implementação de registrador regular $1WMR$, a qual é necessária para a utilização dos protocolos aqui propostos. Assim, um trabalho futuro deve focar na implementação de tal registrador. Por outro lado, vale a pena investir esforço no desenvolvimento de novos algoritmos que utilizem registradores $MWMR$, uma vez que este tipo de registrador serve a muitas aplicações em sistemas dinâmicos baseados em discos.

Um requisito a ser considerado (que não é especificado em (BALDONI; BONOMI; RAYNAL, 2012, 2010a)) é a destruição dos registradores relativos aos processos que deixam o sistema. O tratamento adequado dos eventos de saída certamente melhora o desempenho dos serviços da camada superior.

6.2.2 Desenvolvimento de novos algoritmos com objetos compartilhados mais fortes

Num ambiente de memória compartilhada, uma abordagem que tem sido investigada para a implementação de protocolos fundamentais considera a utilização de objetos mais fortes, com relação ao poder de computabilidade, do que os registradores *read-write*.

Em (HERLIHY, 1991), é introduzido o conceito de *número de consenso* (*consensus number*). Dado um número inteiro n, diz-se que um objeto compartilhado O possui um número de consenso n se existe uma implementação livre de espera para o consenso entre n processos num sistema assíncrono que utiliza apenas objetos do tipo O e registradores *read-write*, mas não existe tal implementação para $n + 1$ processos. Se não existe um n que limite o número de processos, diz-se que o número de consenso do objeto O é $+\infty$. No artigo é mostrado que registradores *read-write* possuem número de consenso 1, enquanto que alguns objetos como *compare&swap* têm número de consenso $+\infty$.

As modernas máquinas multinúcleo trazem consigo o poder do alto desempenho mas também muitos desafios quanto a tolerância a falhas. Essas arquiteturas assíncronas são fortemente influenciadas pelos efeitos da contenção e da hierarquização da memória (HERLIHY; LUCHANGCO, 2008). Elas proveem primitivas atômicas tais como *compare&swap* e *load-linked/store-conditional* que representam a única forma realística de tratar a sincronização nesses sistemas. Assim, pretende-se explorar o desenvolvimento de protocolos fundamentais tais como consenso e eleição de líder para estes ambientes a partir de objetos mais fortes.

6.2.3 Avaliação experimental dos protocolos

Um trabalho que se pretende iniciar em breve é a avaliação dos protocolos propostos através de simulações considerando cenários de interesse. Espera-se que a implementação dos algoritmos acrescente informações sobre sua viabilidade prática.

O desempenho de protocolos executados em sistemas de memória compartilhada envolve basicamente dois aspectos: custo de tempo e custo de espaço. Normalmente, uma avaliação analítica de desempenho dos protocolos considera o número de rodadas/passos executados para que o algoritmo convirja. Essas métricas são fortemente relacionadas ao custo de tempo, uma vez que, grosso modo, quanto maior o número de rodadas/passos, maior o tempo.

Entretanto, é possível que estas métricas não reflitam precisamente o tempo, porque algumas rodadas podem se sobrepor umas às outras, além do que rodadas distintas podem consumir quantidades de tempo distintas. Da mesma forma, o número de acessos (escrita e leitura) concorrentes a um mesmo registrador afeta o tempo de um acesso. No caso de memória compartilhada implementada sobre um sistema de passagem de mensagem, o tempo das operações sobre a memória é afetado pela topologia da rede, a qual determina, por exemplo, quantos saltos são percorridos na rede subjacente para que uma mensagem originada em um determinado nó i alcance determinado nó j. Assim, os custos das diversas mensagens diferem entre si e, consequentemente, os custos de acessos à memória também. Essas diferenças são acentuadas quando a referida rede é dinâmica.

Desta maneira, enquanto numa avaliação analítica os custos das operações de um mesmo tipo sobre a memória (e.g. escritas ou leituras) são considerados iguais, em uma execução real eles podem diferir. Portanto, é desejável que se possa medir tais custos de forma mais precisa. Uma alternativa para se obter tais medidas é através de simulações, de modo que este é também apontado como um trabalho futuro.

REFERÊNCIAS BIBLIOGRÁFICAS

AFEK, Y. et al. Computing with faulty shared objects. *J. ACM*, ACM, New York, NY, USA, v. 42, n. 6, p. 1231–1274, nov. 1995. ISSN 0004-5411. Disponível em: ⟨http://doi.acm.org/10.1145/227683.227688⟩.

AGUILERA, M. et al. Communication-efficient leader election and consensus with limited link synchrony. In: *Proceedings of the Twenty-third Annual ACM Symposium on Principles of Distributed Computing*. New York, NY, USA: ACM, 2004. (PODC '04), p. 328–337. ISBN 1-58113-802-4. Disponível em: ⟨http://doi.acm.org/10.1145/1011767.1011816⟩.

AGUILERA, M. K. A pleasant stroll through the land of infinitely many creatures. *SIGACT News*, ACM, New York, NY, USA, v. 35, n. 2, p. 36–59, jun. 2004. ISSN 0163-5700. Disponível em: ⟨http://doi.acm.org/10.1145/992287.992298⟩.

AGUILERA, M. K. et al. Stable leader election. In: WELCH, J. (Ed.). *Distributed Computing*. Springer Berlin Heidelberg, 2001, (Lecture Notes in Computer Science, v. 2180). p. 108–122. ISBN 978-3-540-42605-9. Disponível em: ⟨http://dx.doi.org/10.1007/3-540-45414-4_8⟩.

AGUILERA, M. K.; ENGLERT, B.; GAFNI, E. On using network attached disks as shared memory. In: *Proceedings of the twenty-second annual symposium on Principles of distributed computing*. New York, NY, USA: ACM, 2003. (PODC '03), p. 315–324. ISBN 1-58113-708-7. Disponível em: ⟨http://doi.acm.org/10.1145/872035.872082⟩.

ALCHIERI, E. A. et al. Byzantine consensus with unknown participants. In: *Proceedings of the 12th International Conference on Principles of Distributed Systems*. Berlin, Heidelberg: Springer-Verlag, 2008. (OPODIS '08), p. 22–40. ISBN 978-3-540-92220-9.

ALON, N. et al. Pragmatic self-stabilization of atomic memory in message-passing systems. In: *Proc. of the Int. Conf. on Stabilization, Safety and Security in Distributed Systems (SSS 2011)*. Grenoble, France: Springer Berlin / Heidelberg, 2011. (Lecture Notes in Computer Science (LNCS)).

ANTA, A. F.; JIMÉNEZ, E.; RAYNAL, M. Eventual leader election with weak assumptions on initial knowledge, communication reliability, and synchrony. In: *In DSN*. [S.l.: s.n.], 2006. p. 166–175.

ANTA, A. F.; JIMÉNEZ, E.; RAYNAL, M. Eventual leader election with weak assumptions on initial knowledge, communication reliability, and synchrony. *Journal of Computer Science and Technology*, Springer US, v. 25, n. 6, p. 1267–1281, 2010. ISSN 1000-9000. Disponível em: ⟨http://dx.doi.org/10.1007/s11390-010-9404-3⟩.

ARANTES, L. et al. Eventual leader election in envolving mobile networks. In: *Proceedings of the 17th International Conference on Principles of Distributed Systems*. Berlin, Heidelberg: Springer-Verlag, 2013. (OPODIS '13).

ASPNES, J. A modular approach to shared-memory consensus, with applications to the probabilistic-write model. *Distributed Computing*, Springer-Verlag, v. 25, n. 2, p. 179–188, 2012. ISSN 0178-2770. Disponível em: ⟨http://dx.doi.org/10.1007/s00446-011-0134-8⟩.

ASPNES, J. *Notes on Theory of Distributed Systems*. Yale University, 2014. Disponível em: ⟨http://cs-www.cs.yale.edu/homes/aspnes/classes/465/notes.pdf⟩.

ATTIYA, H. Efficient and robust sharing of memory in message-passing systems. *J. Algorithms*, v. 34, n. 1, p. 109–127, 2000.

ATTIYA, H.; BAR-NOY, A.; DOLEV, D. Sharing memory robustly in message-passing systems. *J. ACM*, v. 42, n. 1, p. 124–142, 1995.

BALDONI, R. et al. Looking for a definition of dynamic distributed systems. In: MALYSHKIN, V. (Ed.). *Parallel Computing Technologies*. Springer Berlin Heidelberg, 2007, (Lecture Notes in Computer Science, v. 4671). p. 1–14. ISBN 978-3-540-73939-5. Disponível em: ⟨http://dx.doi.org/10.1007/978-3-540-73940-1_1⟩.

BALDONI, R. et al. Implementing a register in a dynamic distributed system. In: *Distributed Computing Systems, 2009. ICDCS '09. 29th IEEE International Conference on*. [S.l.: s.n.], 2009. p. 639–647. ISSN 1063-6927.

BALDONI, R.; BONOMI, S.; RAYNAL, M. Joining a distributed shared memory computation in a dynamic distributed system. In: *Proc. 7th Workshop on Software Technologies for Future Embedded and Ubiquitous Computing Systems*. [S.l.]: Springer-Verlag LNCS, 2009. (SEUS'09), p. 91–102.

BALDONI, R.; BONOMI, S.; RAYNAL, M. Regular register: An implementation in a churn prone environment. In: KUTTEN, S.; Å½EROVNIK, J. (Ed.). *Structural Information and Communication Complexity*. Springer Berlin Heidelberg, 2010, (Lecture Notes in Computer Science, v. 5869). p. 15–29. ISBN 978-3-642-11475-5. Disponível em: ⟨http://dx.doi.org/10.1007/978-3-642-11476-2_3⟩.

BALDONI, R.; BONOMI, S.; RAYNAL, M. Value-based sequential consistency for set objects in dynamic distributed systems. In: DâAMBRA, P.; GUARRACINO, M.; TALIA, D. (Ed.). *Euro-Par 2010 - Parallel Processing*. Springer Berlin Heidelberg, 2010, (Lecture Notes in Computer Science, v. 6271). p. 523–534. ISBN 978-3-642-15276-4. Disponível em: ⟨http://dx.doi.org/10.1007/978-3-642-15277-1_50⟩.

BALDONI, R.; BONOMI, S.; RAYNAL, M. Implementing a regular register in an eventually synchronous distributed system prone to continuous churn. *Parallel and Distributed Systems, IEEE Transactions on*, v. 23, n. 1, p. 102–109, Jan 2012. ISSN 1045-9219.

BLOND, S. L.; FESSANT, F. L.; MERRER, E. L. Finding good partners in availability-aware p2p networks. In: GUERRAOUI, R.; PETIT, F. (Ed.). *Stabilization, Safety, and Security of Distributed Systems*. Springer Berlin Heidelberg, 2009, (Lecture Notes in Computer Science, v. 5873). p. 472–484. ISBN 978-3-642-05117-3. Disponível em: ⟨http://dx.doi.org/10.1007/978-3-642-05118-0_33⟩.

CACHIN, C.; GUERRAOUI, R.; RODRIGUES, L. *Introduction to Reliable and Secure Distributed Programming*. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-15260-3. Disponível em: ⟨http://www.springer.com/computer/theoretical+computer+science/book/978-3-642-15259-7⟩.

CAVIN, D.; SASSON, Y.; SCHIPER, A. Consensus with unknown participants or fundamental self-organization. In: *Proc. of the 3rd Int. Conf. on AD-NOC Networks & Wireless (ADHOC-NOW'04)*. Vancouver: Springer-Verlag, 2004. p. 135–148.

CAVIN, D.; SASSON, Y.; SCHIPER, A. *Reaching Agreement with Unknown Participants in Mobile Self-organized Networks in Spite of Process Crashes*. [S.l.], 2005.

CHANDRA, T.; TOUEG, S. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, v. 43, n. 2, p. 225–267, mar. 1996.

CHANDRA, T. D.; HADZILACOS, V.; TOUEG, S. The weakest failure detector for solving consensus. *Journal of the ACM*, v. 43, n. 4, p. 685–722, jul. 1996.

CHOCKLER, G.; MALKHI, D. Active disk paxos with infinitely many processes. In: *Proceedings of the twenty-first annual symposium on Principles of distributed computing*. New York, NY, USA: ACM, 2002. (PODC '02), p. 78–87. ISBN 1-58113-485-1. Disponível em: ⟨http://doi.acm.org/10.1145/571825.571837⟩.

COULOURIS, G. et al. *Distributed Systems: Concepts and Design*. [S.l.]: Addison-Wesley, 2011.

CRISTIAN, F. Understanding fault-tolerant distributed systems. *Commun. ACM*, ACM, New York, NY, USA, v. 34, n. 2, p. 56–78, fev. 1991. ISSN 0001-0782. Disponível em: ⟨http://doi.acm.org/10.1145/102792.102801⟩.

CRISTIAN, F.; FETZER, C. The timed asynchronous distributed system model. *Parallel and Distributed Systems, IEEE Transactions on*, v. 10, n. 6, p. 642–657, Jun 1999. ISSN 1045-9219.

DELPORTE-GALLET, C.; FAUCONNIER, H. Two consensus algorithms with atomic registers and failure detector omega. In: *Proceedings of the 10th International Conference on Distributed Computing and Networking*. Berlin, Heidelberg: Springer-Verlag, 2009. (ICDCN '09), p. 251–262. ISBN 978-0-7695-3659-0.

DELPORTE-GALLET, C. et al. 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. New York, NY, USA: ACM, 2004. (PODC '04), p. 338–346. ISBN 1-58113-802-4. Disponível em: ⟨http://doi.acm.org/10.1145/1011767.1011818⟩.

DOLEV, D.; DWORK, C.; STOCKMEYER, L. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, ACM, New York, NY, USA, v. 34, p. 77–97, January 1987. ISSN 0004-5411. Disponível em: ⟨http://doi.acm.org/10.1145/7531.7533⟩.

DOLEV, S. et al. Crash resilient and pseudo-stabilizing atomic registers. In: *Proceedings of the International Conference on Principles of Distributed Systems (OPODIS 2012)*. Rome, Italy: Springer Berlin / Heidelberg, 2012. (Lecture Notes in Computer Science (LNCS)).

DUTTA, P.; GUERRAOUI, R. Fast indulgent consensus with zero degradation. In: *Proceedings of the 4th European Dependable Computing Conference on Dependable Computing*. London, UK, UK: Springer-Verlag, 2002. (EDCC-4), p. 191–208. ISBN 3-540-00012-7. Disponível em: ⟨http://dl.acm.org/citation.cfm?id=645333.649854⟩.

DWORK, C.; LYNCH, N.; STOCKMEYER, L. Consensus in the presence of partial synchrony. *Journal of ACM*, ACM, New York, NY, USA, v. 35, p. 288–323, April 1988. ISSN 0004-5411. Disponível em: ⟨http://doi.acm.org/10.1145/42282.42283⟩.

FERNÁNDEZ, A.; JIMÉNEZ, E.; RAYNAL, M. Electing an eventual leader in an asynchronous shared memory system. In: *Dependable Systems and Networks, 2007. DSN '07. 37th Annual IEEE/IFIP International Conference on*. [S.l.: s.n.], 2007. p. 399–408.

FERNÁNDEZ, A. et al. A timing assumption and two t-resilient protocols for implementing an eventual leader service in asynchronous shared memory systems. *Algorithmica*, Springer-Verlag, v. 56, n. 4, p. 550–576, 2010. ISSN 0178-4617. Disponível em: ⟨http://dx.doi.org/10.1007/s00453-008-9190-2⟩.

FISCHER, M. J.; LYNCH, N. A.; PATERSON, M. D. Impossibility of distributed consensus with one faulty process. *Journal of ACM*, v. 32, n. 2, p. 374–382, abr. 1985.

GAFNI, E.; LAMPORT, L. Disk paxos. *Distrib. Comput.*, Springer-Verlag, London, UK, UK, v. 16, n. 1, p. 1–20, fev. 2003. ISSN 0178-2770. Disponível em: ⟨http://dx.doi.org/10.1007/s00446-002-0070-8⟩.

GAFNI, E.; MERRITT, M.; TAUBENFELD, G. The concurrency hierarchy, and algorithms for unbounded concurrency. In: *Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing*. New York, NY, USA: ACM, 2001. (PODC '01), p. 161–169. ISBN 1-58113-383-9. Disponível em: ⟨http://doi.acm.org/10.1145/383962.384008⟩.

GODFREY, P. B.; SHENKER, S.; STOICA, I. Minimizing churn in distributed systems. In: *Proceedings of the 2006 Conference on Applications, Technologies, Architectures,*

and Protocols for Computer Communications. New York, NY, USA: ACM, 2006. (SIGCOMM '06), p. 147–158. ISBN 1-59593-308-5. Disponível em: ⟨http://doi.acm.org/10.1145/1159913.1159931⟩.

GOMEZ-CALZADO, C. et al. Fault-tolerant leader election in mobile dynamic distributed systems. In: *Dependable Computing (PRDC), 2013 IEEE 19th Pacific Rim International Symposium on*. [S.l.: s.n.], 2013. p. 78–87.

GREVE, F.; TIXEUIL, S. Knowledge connectivity *vs.* synchrony requirements for fault-tolerant agreement in unknown networks. In: *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*. Washington, USA: IEEE Computer Society, 2007. p. 82–91.

GREVE, F.; TIXEUIL, S. Conditions for the solvability of fault-tolerant consensus in asynchronous unknown networks: Invited paper. In: *III ACM SIGACT-SIGOPS International Workshop on Reliability, Availability, and Security. Co-located with the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*. [S.l.: s.n.], 2010.

GUERRAOUI, R. Indulgent algorithms. In: *Proc. 19^{th} ACM Symp. on Principles of Distributed Computing (PODC)*. Portland: [s.n.], 2000. p. 289–298.

GUERRAOUI, R.; KAPALKA, M.; KOUZNETSOV, P. The weakest failure detectors to boost obstruction-freedom. In: DOLEV, S. (Ed.). *Distributed Computing*. Springer Berlin Heidelberg, 2006, (Lecture Notes in Computer Science, v. 4167). p. 399–412. ISBN 978-3-540-44624-8. Disponível em: ⟨http://dx.doi.org/10.1007/11864219_28⟩.

GUERRAOUI, R.; LEVY, R. R. Robust emulations of shared memory in a crash-recovery model. *2012 IEEE 32nd International Conference on Distributed Computing Systems*, IEEE Computer Society, Los Alamitos, CA, USA, v. 0, p. 400–407, 2004. ISSN 1063-6927.

GUERRAOUI, R.; RAYNAL, M. The information structure of indulgent consensus. *Computers, IEEE Transactions on*, v. 53, n. 4, p. 453–466, April 2004. ISSN 0018-9340.

GUERRAOUI, R.; RAYNAL, M. A leader election protocol for eventually synchronous shared memory systems. In: *Software Technologies for Future Embedded and Ubiquitous Systems, 2006 and the 2006 Second International Workshop on Collaborative Computing, Integration, and Assurance. SEUS 2006/WCCIA 2006. The Fourth IEEE Workshop on*. [S.l.: s.n.], 2006. p. 6 pp.–.

GUERRAOUI, R.; RAYNAL, M. The alpha of indulgent consensus. *The Computer Journal*, v. 50, n. 1, p. 53–67, 2007. Disponível em: ⟨http://comjnl.oxfordjournals.org/content/50/1/53.abstract⟩.

GUERRAOUI, R.; SCHIPER, A. The generic consensus service. *Software Engineering, IEEE Transactions on*, v. 27, n. 1, p. 29–41, Jan 2001. ISSN 0098-5589.

GUMMADI, K. P. et al. Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In: *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*. New York, NY, USA: ACM, 2003. (SOSP '03), p. 314–329. ISBN 1-58113-757-5. Disponível em: ⟨http://doi.acm.org/10.1145/945445.945475⟩.

HADZILACOS, V.; TOUEG, S. *A Modular Approach to Fault-tolerant Broadcasts and Related Problems*. [S.l.], 1994.

HERLIHY, M. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, ACM, New York, NY, USA, v. 13, n. 1, p. 124–149, jan. 1991. ISSN 0164-0925. Disponível em: ⟨http://doi.acm.org/10.1145/114005.102808⟩.

HERLIHY, M.; LUCHANGCO, V. Distributed computing and the multicore revolution. *SIGACT News*, ACM, New York, NY, USA, v. 39, n. 1, p. 62–72, mar. 2008. ISSN 0163-5700. Disponível em: ⟨http://doi.acm.org/10.1145/1360443.1360458⟩.

HERLIHY, M.; LUCHANGCO, V.; MOIR, M. Obstruction-free synchronization: Double-ended queues as an example. In: *Proceedings of the 23rd International Conference on Distributed Computing Systems*. Washington, DC, USA: IEEE Computer Society, 2003. (ICDCS '03), p. 522–. ISBN 0-7695-1920-2. Disponível em: ⟨http://dl.acm.org/citation.cfm?id=850929.851942⟩.

HERLIHY, M. et al. Software transactional memory for dynamic-sized data structures. In: *Proceedings of the Twenty-second Annual Symposium on Principles of Distributed Computing*. New York, NY, USA: ACM, 2003. (PODC '03), p. 92–101. ISBN 1-58113-708-7. Disponível em: ⟨http://doi.acm.org/10.1145/872035.872048⟩.

HERLIHY, M.; RAJSBAUM, S.; RAYNAL, M. Power and limits of distributed computing shared memory models. *Theor. Comput. Sci.*, v. 509, p. 3–24, 2013. Disponível em: ⟨http://dx.doi.org/10.1016/j.tcs.2013.03.002⟩.

HERLIHY, M. P. Impossibility and universality results for wait-free synchronization. In: *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*. New York, NY, USA: ACM, 1988. (PODC '88), p. 276–290. ISBN 0-89791-277-2. Disponível em: ⟨http://doi.acm.org/10.1145/62546.62593⟩.

HERLIHY, M. P.; WING, J. M. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, ACM, New York, NY, USA, v. 12, n. 3, p. 463–492, jul 1990. ISSN 0164-0925. Disponível em: ⟨http://doi.acm.org/10.1145/78969.78972⟩.

HUTLE, M. et al. Chasing the weakest system model for implementing ω and consensus. *IEEE Transactions on Dependable and Secure Computing*, IEEE Computer Society, Los Alamitos, CA, USA, v. 6, n. 4, p. 269–281, 2009. ISSN 1545-5971.

JAYANTI, P.; CHANDRA, T.; TOUEG, S. Fault-tolerant wait-free shared objects. In: *Foundations of Computer Science, 1992. Proceedings., 33rd Annual Symposium on*. [S.l.: s.n.], 1992. p. 157–166.

JIMÉNEZ, E.; ARÉVALO, S.; FERNÁNDEZ, A. Implementing unreliable failure detectors with unknown membership. *Information Processing Letters*, v. 100, n. 2, p. 60 – 63, 2006. ISSN 0020-0190. Disponível em: ⟨http://www.sciencedirect.com/science/article/pii/S0020019006001621⟩.

KHOURI, C.; GREVE, F. Algoritmo de consenso genérico em memória compartilhada. In: . Brasília, DF, Brasil: SBC, 2013. (XIV WTF-SBRC).

KHOURI, C.; GREVE, F. A generic consensus algorithm for shared memory. In: *The 19th IEEE Pacific Rim International Symposium on Dependable Computing*. Vancouver: [s.n.], 2013. (PRDC 2013).

KHOURI, C.; GREVE, F. Servico de líder em sistemas dinâmicos com memória compartilhada. In: . Florianópolis, SC, Brasil: SBC, 2014. (XIV WTF-SBRC).

KHOURI, C.; GREVE, F.; TIXEUIL, S. Consensus with unknown participants in shared memory. In: *Reliable Distributed Systems (SRDS), 2013 IEEE 32nd International Symposium on*. New York, NY, USA: [s.n.], 2013. p. 51–60.

KO, S.; HOQUE, I.; GUPTA, I. Using tractable and realistic churn models to analyze quiescence behavior of distributed protocols. In: *Reliable Distributed Systems, 2008. SRDS '08. IEEE Symposium on*. [S.l.: s.n.], 2008. p. 259–268. ISSN 1060-9857.

KONG, J.; BRIDGEWATER, J.; ROYCHOWDHURY, V. A general framework for scalability and performance analysis of dht routing systems. In: *Dependable Systems and Networks, 2006. DSN 2006. International Conference on*. [S.l.: s.n.], 2006. p. 343–354.

LAMPORT, L. On interprocess communication. *Distributed Computing*, v. 1, n. 2, p. 77–101, 1986.

LAMPORT, L. The part-time parliament. *ACM Transactions on Computer Systems*, v. 16, n. 2, p. 133–169, maio 1998.

LARREA, M. et al. Specifying and implementing an eventual leader service for dynamic systems. *Int. J. Web Grid Serv.*, Inderscience Publishers, Inderscience Publishers, Geneva, SWITZERLAND, v. 8, n. 3, p. 204–224, set. 2012. ISSN 1741-1106. Disponível em: ⟨http://dx.doi.org/10.1504/IJWGS.2012.049167⟩.

LO, W.-K.; HADZILACOS, V. Using failure detectors to solve consensus in asynchronous shared-memory systems (extended abstract). In: *Proceedings of the 8th International Workshop on Distributed Algorithms*. London, UK: Springer-Verlag, 1994. (WDAG '94), p. 280–295. ISBN 3-540-58449-8. Disponível em: ⟨http://dl.acm.org/citation.cfm?id=645951.675468⟩.

LOUI, M. C.; ABU-AMARA, H. H. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, JAI Press, v. 4, p. 163–183, 1987.

MALKHI, D.; OPREA, F.; ZHOU, L. ω meets paxos: Leader election and stability without eventual timely links. In: *Proceedings of the 19th International Conference on Distributed Computing*. Berlin, Heidelberg: Springer-Verlag, 2005. (DISC'05), p. 199–213. ISBN 3-540-29163-6, 978-3-540-29163-3. Disponível em: ⟨http://dx.doi.org/10.1007/11561927_16⟩.

MERRITT, M.; TAUBENFELD, G. Computing with infinitely many processes. *Information and Computation*, v. 233, n. 0, p. 12 – 31, 2013. ISSN 0890-5401. Disponível em: ⟨http://www.sciencedirect.com/science/article/pii/S0890540113001314⟩.

MOSTEFAOUI, A.; MOURGAYA, E.; RAYNAL, M. M.: Asynchronous implementation of failure detectors. In: *In: Proc. International IEEE Conference on Dependable Systems and Networks (DSN'03*. [S.l.: s.n.], 2003. p. 351–360.

MOSTEFAOUI, A. et al. A time-free assumption to implement eventual leadership. *Parallel Processing Letters*, v. 16, n. 02, p. 189–207, 2006. Disponível em: ⟨http://www.worldscientific.com/doi/abs/10.1142/S0129626406002575⟩.

MOSTEFAOUI, A.; RAYNAL, M. Leader-based consensus. *IRISA - Publication interne*, IRISA, n. 1372, p. 1–13, Decembre 2000. Disponível em: ⟨courses.csail.mit.edu/6.895/fall02/papers/Mostefaoui/PI-1372.ps.gz⟩.

MOSTEFAOUI, A.; RAYNAL, M.; TRAVERS, C. Crash-resilient time-free eventual leadership. In: *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems*. Washington, DC, USA: IEEE Computer Society, 2004. (SRDS '04), p. 208–217. ISBN 0-7695-2239-4. Disponível em: ⟨http://dl.acm.org/citation.cfm?id=1032662.1034366⟩.

MOSTEFAOUI, A. et al. From static distributed systems to dynamic systems. In: *Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems*. [S.l.]: IEEE Pres, 2005. (SRDS'05), p. 109–118. ISBN 0-7695-2463-X.

NISHIMURA, N. Asynchronous shared memory parallel computation. In: *Proceedings of the Second Annual ACM Symposium on Parallel Algorithms and Architectures*. New York, NY, USA: ACM, 1990. (SPAA '90), p. 76–84. ISBN 0-89791-370-1. Disponível em: ⟨http://doi.acm.org/10.1145/97444.97672⟩.

Paradigms for distributed programs. In: PAUL, M. et al. (Ed.). *Distributed Systems*. Springer Berlin Heidelberg, 1985, (Lecture Notes in Computer Science, v. 190). p. 431–480. ISBN 978-3-540-15216-3. Disponível em: ⟨http://dx.doi.org/10.1007/3-540-15216-4_18⟩.

RAYNAL, M.; STAINER, J. A simple asynchronous shared memory consensus algorithm based on omega and closing sets. In: *Complex, Intelligent and Software Intensive Systems (CISIS), 2012 Sixth International Conference on*. [S.l.: s.n.], 2012. p. 357–364.

TANENBAUM, A. S.; STEEN, M. V. *Distributed Systems: Principles and Paradigms*. [S.l.]: Pearson Prentice Hall, 2006.

TOP500. Top500 list. 2014. Disponível em: ⟨http://www.top500.org/lists/2014/11/⟩.

YELLEN, J.; GROSS, J. *Graph Theory and Its Applications*. [S.l.]: CRC Press, 1998.