Detectando Falhas Bizantinas em Sistemas Distribuídos Dinâmicos*

Murilo Santos de Lima †

Departamento de Ciência da Computação
Instituto de Matemática e Estatística
Universidade de São Paulo (USP)
Rua do Matão, 1010, Cidade Universitária
05508-090 – São Paulo – SP – Brazil
mslima@ime.usp.br

Fabíola Gonçalves Pereira Greve

Departamento de Ciência da Computação
Instituto de Matemática
Universidade Federal da Bahia (UFBA)
Av. Adhemar de Barros, S/N, Campus de Ondina
40170-110 – Salvador – BA – Brazil
fabiola@dcc.ufba.br

Resumo

Em um sistema distribuído dinâmico, no qual os nós podem entrar e sair da rede aleatoriamente, o desafio de implementar serviços confiáveis é grande. Nesse contexto, um fator preocupante, em especial, é a segurança. Detectores de falhas bizantinas (ou arbitrárias) são uma solução elegante para problemas de segurança, uma vez que separam o tratamento das falhas do protocolo distribuído que os utiliza. No entanto, desconhecem-se trabalhos na literatura descrevendo soluções específicas para sistemas dinâmicos. Este artigo propõe um detector de falhas bizantinas para tais sistemas. Adicionalmente, o protocolo apresentado é assíncrono, isto é, não se baseia no uso de temporizadores para a detecção das falhas, o que favorece sua escalabilidade e adaptabilidade.

Palavras-chave: detectores de falhas, falhas bizantinas, sistemas distribuídos dinâmicos, sistemas auto-organizáveis, detectores de falhas assíncronos

Abstract

Byzantine failure detectors provide an elegant abstraction for solving security problems. However, as far as we know, there is no complete solution for this problem in a dynamic distributed system. This paper presents thus a first Byzantine failure detector for this context. The protocol has the interesting feature to be asynchronous, that is, the failure detection process does not rely on timers to make suspicions. This characteristic favors its scalability and adaptability.

Keywords: failure detectors, Byzantine failures, dynamic distributed systems, self-organizing systems, asynchronous failure detectors

1. Introdução

Um sistema distribuído dinâmico [1] é composto por uma população de nós[1] que podem entrar e sair da rede aleatoriamente, a qualquer momento da execução do sistema. Comumente, as seguintes características são apresentadas: (1) a latência na transmissão das mensagens é imprevisível, (2) a rede não está totalmente conectada, (3) não existem elementos centralizadores na rede, (4) não é possível prover aos processos uma visão global da topologia da rede, de forma que cada nó tem um conhecimento parcial da composição do sistema e (5) os nós podem alterar sua posição quanto à topologia da rede. Constata-se, portanto, que os protocolos distribuídos clássicos, que supõem uma rede com composição estática e conhecida, não são mais adequados neste novo contexto, característico das redes entre pares (P2P), redes móveis auto-organizáveis (Manets) e redes de sensores sem fio.

Um fator preocupante em sistemas distribuídos dinâmicos, em especial, é a segurança. A dinamicidade da população dos nós e o uso comum de redes sem fio ou da Internet como meios de comunicação facilitam a ação de agentes maliciosos sobre o sistema. O modelo de falhas bizantinas [2] lida com este tipo de ataque ao considerar a existência de processos corruptos, que podem se comportar de maneira arbitrária na tentativa de impedir que o sistema funcione conforme a sua especificação. Um processo bizantino pode, por exemplo, tentar assumir

*Uma versão preliminar deste trabalho foi apresentada no SBRC 2009. O trabalho tem apoio da FAPESB-Bahia e do CNPq-Brasil.

†Murilo é Bacharel em Ciência da Computação pela Universidade Federal da Bahia e este trabalho é resultado do seu projeto de final de curso. Parte deste trabalho foi desenvolvido por Murilo no Instituto Recôncavo de Tecnologia – Salvador – Bahia – Brasil.

[1]Neste trabalho, os termos "nó" e "processo" serão utilizados indistintamente.

a identidade de outro, enviar mensagens com valores incorretos, duplicar mensagens ou simplesmente não enviar mensagens que o protocolo em execução especificar. O desenvolvimento de sistemas tolerantes a faltas bizantinas [3], isto é, que mantenham o seu funcionamento correto apesar do comportamento maligno de alguns de seus processos, é portanto de especial interesse.

A abstração de detectores de falhas [4] provê uma forma modular de tratar as falhas em sistemas assíncronos, isto é, sistemas que não atendam a restrições temporais. O detector separa o tratamento das falhas e os requisitos de sincronia do protocolo distribuído que o utiliza, de forma que este pode lidar apenas com a tarefa a que se propõe. O problema do consenso [5], por exemplo, que sumariza diversos problemas de acordo distribuído, não pode ser resolvido num sistema sem quaisquer suposições de sincronia, mesmo se apenas um processo falhar por colapso (*crash*) [6]. Se o sistema dispõe, no entanto, de um detector da classe $\Diamond\mathcal{S}$ [4], o consenso pode ser resolvido para falhas por colapso sem lidar diretamente com questões de sincronia.

A grande maioria das implementações para detectores de falhas considera a existência de uma rede estática, cuja composição dos nós é conhecida [7]. Mais recentemente, algumas propostas foram feitas para sistemas móveis e auto-organizáveis. Em [8], Gupta *et al.* consideram uma população dinâmica dos nós; já Friedman e Tcharny [9] toleram a mobilidade dos nós. Todas essas propostas utilizam um mecanismo de temporização para detecção das falhas, i.e., os nós monitoram a vivacidade dos demais nós da rede através da troca de mensagens do tipo "eu estou vivo". Esse mecanismo, no entanto, não se adéqua muito bem a sistemas distribuídos dinâmicos, devido à imprevisibilidade na latência da transmissão das mensagens.

Detectores de falhas assíncronos não fazem uso do recurso de temporização. Nessa linha, um trabalho pioneiro foi apresentado por Mostefaoui *et al.* [10]. Entretanto, ele considera uma rede estática com composição dos nós conhecida. Em [11], Sens *et al.* apresentam uma implementação assíncrona de um detector de falhas que trata tanto a dinamicidade da composição da rede quanto a mobilidade dos nós. Todos esses trabalhos, no entanto, são focados na detecção de falhas por colapso, i.e., quando um processo se comporta de maneira benigna até falhar e haver o término da execução.

Alguns trabalhos [12, 13] são voltados para detecção de falhas bizantinas. Entretanto, nenhum deles lida com a dinamicidade do sistema e mobilidade dos seus nós. Além disso, todos fundamentam-se no uso de temporizadores. Os autores desconhecem, portanto, trabalhos que proponham detectores de falhas bizantinas específicos para sistemas distribuídos dinâmicos e também desconhecem detectores de falhas bizantinas que sejam assíncronos. Existem trabalhos voltados para problemas específicos, como é o caso de Awerbuch *et al.* [14], que resolvem a detecção para o roteamento, ou então, o de Aguilera *et al.* [15] que só toleram falhas de omissão e não consideram falhas de segurança.

Este artigo apresenta, então, o primeiro protocolo para detecção de falhas bizantinas em sistemas dinâmicos e que, além disso, tem a característica de ser assíncrono. O protocolo tem por base as abordagens descritas por Kihlstrom *et al.* [12] para falhas bizantinas e Sens *et al.* [11] para a detecção assíncrona. Assim, as falhas de segurança são detectadas através da definição de um formato preestabelecido para as mensagens trocadas pelos processos. Estas devem ser autenticadas e seu conteúdo certificado. Quanto às falhas de omissão, a detecção se baseia no padrão de mensagens imposto pelo algoritmo que utiliza o detector, o que é razoável, uma vez que as falhas bizantinas são definidas como desvio do comportamento especificado pelo algoritmo [16]. A detecção assíncrona é possível graças ao padrão de troca de mensagens, à topologia da rede e a certas propriedades comportamentais seguidas pelos processos no sistema. O protocolo não tolera, entretanto, a mobilidade dos nós.

A detecção assíncrona, na qual o protocolo se baseia, evidenciou dois importantes resultados. Primeiro, conjectura-se ser impossível projetar algoritmos distribuídos assíncronos tolerantes a falhas bizantinas sem que o padrão de troca de mensagens entre os processos seja distribuído. Além disso, conclui-se que o uso da difusão local em canais confiáveis, comum em redes sem fio, simplifica o tratamento das falhas de segurança, de forma que determinadas classes de falhas não se configuram no sistema.

O restante deste artigo encontra-se estruturado da seguinte forma: na Seção 2, define-se o modelo de sistema. Uma caracterização do modelo de falhas bizantinas é apresentada na Seção 3. As Seções 4-6 apresentam o protocolo desenvolvido, as propriedades comportamentais que o sistema deve atender para que o protocolo funcione corretamente e sua implementação. A prova da sua correção é exibido na Seção 7. Por fim, na Seção 8 apresentam-se as conclusões e propostas de trabalhos futuros.

2. MODELO DE SISTEMA

Supõe-se um sistema distribuído composto por um conjunto $\Pi = \{p_1, p_2, \ldots, p_n\}$ com $n > 4$ processos. Nós podem entrar e sair da rede aleatoriamente, a qualquer momento da execução do sistema. Não são feitas quaisquer restrições sobre a velocidade dos processadores, sobre o desvio relativo dos relógios ou sobre os atrasos de transmissão das mensagens; isto é, supõe-se um sistema *assíncrono*. Supõe-se a ocorrência de falhas bi-

zantinas: os processos podem falhar de maneira arbitrária, inclusive maliciosa; supõe-se entretanto a existência de um mecanismo de autenticação de mensagens. Os processos não têm conhecimento de Π ou n, apenas de um subconjunto de Π com os quais estabeleceram comunicação anteriormente. O número máximo de falhas tolerado f é conhecido por todos os processos. Supõe-se, para simplificação das definições e demonstrações apresentadas, a existência de um tempo global t, embora o mesmo não seja conhecido pelos processos. Cada processo é identificado unicamente e processos faltosos não podem obter mais de um identificador, de forma que é impossível a ocorrência de *ataques sybil* [17].

Os processos trocam mensagens por difusão local (*broadcast*) através de uma rede de comunicação sem fio. Supõe-se a existência de canais locais confiáveis, de forma que as mensagens difundidas são recebidas em algum momento no futuro por todos os processos corretos dentro da área de cobertura (*range*) do transmissor. Os canais não duplicam, alteram ou inserem novas mensagens e todo processo correto autentica suas mensagens de forma incorruptível. Implementações de comunicação por difusão em canais confiáveis para redes sem fio encontram-se em [18, 19].

O sistema pode ser representado por um grafo não-direcionado $G(V, E)$, sendo $V = \Pi$ e $\{p_i, p_j\} \in E$ se p_i e p_j encontram-se no *range* um do outro (p_i e p_j são denominados *vizinhos*). A área de cobertura $range_i$ de um nó p_i é definida pelo conjunto $range_i := \{p_j \in \Pi : (p_i, p_j) \in E\}$. Note-se $range_i$ corresponde à vizinhança de p_i em G, $|range_i|$ equivale ao grau de p_i em G e que $p_i \in range_j \Leftrightarrow p_j \in range_i$. Ou seja, a comunicação entre os processos é *simétrica*.

Definição 1 (Densidade da área de cobertura (d)) *A densidade d de um grafo de comunicação $G(V, E)$ consiste no tamanho do menor range da rede, isto é: $d := min\ \{|range_i| : i \in \{1, 2, ..., n\}\}$. d é portanto o grau mínimo do grafo G.*

Considera-se que o parâmetro d da rede é conhecido por todos os processos.

Definição 2 (Rede com f-cobertura bizantina) *Uma rede de comunicação representada pelo grafo $G(V, E)$ tem f-cobertura bizantina se e somente se G é $(f + 1)$-conexo e $d \geq 2f + 1$.*

Um grafo k-conexo possui k caminhos distintos nos vértices entre quaisquer par de vértices, o que leva à seguinte observação:

Observação 1 *Em uma rede com f-cobertura bizantina, apesar da presença de $f < n$ processos faltosos, existirá pelo menos um caminho formado apenas por nós corretos entre cada par de nós corretos.*

3. FALHAS BIZANTINAS

A identificação das falhas bizantinas é caracterizada pela satisfação de dois requisitos: (1) os processos corretos devem possuir uma visão coerente das mensagens enviadas por cada processo e (2) os processos corretos devem poder verificar se as mensagens enviadas pelos demais processos estão consistentes com os requisitos do algoritmo sendo executado, o que evidencia que a detecção de falhas bizantinas é definida em função de determinado algoritmo ou protocolo. O primeiro requisito pode ser atendido utilizando duas técnicas distintas: a *redundância da informação* e o uso de *assinaturas digitais não-forjáveis* [2]. O segundo, pela adição de informação às mensagens, na forma de *certificados*, que possam ser utilizados para validar o conteúdo sendo transmitido [12].

A Figura 1 ilustra as classes de falhas bizantinas descritas por Kihlstrom *et al.* [12], distinguindo-se duas superclasses: *detectáveis*, quando comportamento externo do processo faltoso fornece evidências de que o mesmo falhou e *não-detectáveis*, caso contrário. Falhas não-detectáveis podem ser subdivididas em *não-observáveis*, quando os demais processos não podem notar a ocorrência da falha (por exemplo, um processo faltoso informa um parâmetro fornecido pelo usuário incorretamente) e *não-diagnosticáveis*, quando não é possível identificar o processo que gerou a falha (por exemplo, os processos recebem uma mensagem não assinada).

As falhas detectáveis são classificadas em falhas *de progresso* (ou falhas por *omissão*) e falhas *de segurança* (ou falhas por *comissão*). Falhas de progresso atrapalham a terminação da computação, uma vez que o processo faltoso não envia mensagens requeridas por sua especificação ou as envia a apenas parte dos processos do sistema. Falhas de segurança violam propriedades invariantes às quais os processos devem atender, podendo ser definidas como o não-cumprimento de uma das seguintes restrições: (1) um processo deve enviar as mesmas mensagens para todos os outros (um processo faltoso poderia, portanto, enviar a mesma mensagem com valores distintos para processos distintos); (2) as mensagens enviadas devem estar de acordo com o algoritmo sendo executado.

Propriedades do detector de falhas. Kihlstrom *et al.* [12] também definem classes de detectores de falhas bizantinas que diferem das descritas por Chandra e Toueg [4], uma vez que ali são detectadas apenas falhas por colapso. Seja $\mathcal{A}$ um algoritmo que utiliza o detector de falhas como módulo subjacente. A classe $\Diamond\mathcal{S}(Byz, \mathcal{A})$, que é o foco deste trabalho, é definida pelas seguintes propriedades:

(i) *Completude forte bizantina* (para algoritmo $\mathcal{A}$): em algum momento no futuro todo processo correto suspeita permanentemente de todo processo que se desviou de $\mathcal{A}$

Figura 1. Categorização das falhas bizantinas [12]

de forma detectável;
(ii) *Precisão fraca após um tempo*: em algum momento no futuro, um processo correto não será suspeito por qualquer processo correto.

4. Princípios básicos do protocolo de detecção assíncrona de falhas bizantinas

Nesta seção são apresentados os princípios fundamentais para o projeto do detector assíncrono para falhas bizantinas.

4.1. Padrão de troca de mensagens

Grande parte dos protocolos de detecção de falhas por colapso baseiam-se num mecanismo de troca de mensagens do tipo "eu estou vivo" (*I'm alive*). Entretanto, no modelo bizantino, devido à ocorrência de processos maliciosos, tal mecanismo não é mais suficiente. Um processo faltoso pode responder corretamente às mensagens do detector de falhas sem no entanto garantir o progresso e a segurança do algoritmo sendo executado. Conseqüentemente, a detecção das falhas deve se basear no padrão das mensagens enviadas na execução do algoritmo $\mathcal{A}$ que utiliza o detector de falhas. Assim sendo, de forma similar ao feito por Kihlstrom *et al.* [12], as suspeitas são levantadas em função das mensagens requeridas por $\mathcal{A}$. Entretanto, neste trabalho tais suspeitas são identificadas de maneira assíncrona, sem recorrer ao uso de temporizadores (mecanismos de *timeout*) para detectar falhas de omissão. Além disso, a detecção segue um padrão de troca de mensagens local, ou seja, entre os nós que se encontram na vizinhança.

A detecção assíncrona das falhas é feita aguardando-se a recepção de determinada mensagem a partir de $(d - f)$ remetentes distintos. $(d - f)$ corresponde à quantidade mínima de nós corretos na vizinhança de determinado nó na rede. Esta estratégia é a mesma seguida por Sens *et al.* [11]. Entretanto, o padrão de mensagens QUERY-RESPONSE, no qual eles se fundamentam, não é adequado ao modelo bizantino, pois, como dito anteriormente, um processo faltoso pode enviar mensagens RESPONSE sem no entanto atender aos requisitos de progresso do algoritmo $\mathcal{A}$. O protocolo aqui apresentado irá, portanto, efetuar as suas suspeitas com base na troca de mensagens requeridas por $\mathcal{A}$. Assim, propõe-se que os processos aguardem a recepção das mensagens de $\mathcal{A}$ a partir de pelo menos $(d - f)$ remetentes distintos, suspeitando-se da omissão dos demais processos. É importante observar que para que tal suspeita possa ser feita, o padrão de comunicação do algoritmo $\mathcal{A}$ deve ser distribuído. Ou seja, em cada passo, todos processos devem trocar mensagens entre si, seguindo o padrão $n \rightarrow n$. Protocolos com esse padrão são denominados "simétricos". Como o detector segue um padrão local de troca de mensagens, essa comunicação deve ocorrer pelo menos entre os processos que estão na mesma vizinhança. Algoritmos de consenso simétricos que utilizam um detector de falhas $\Diamond\mathcal{S}$ foram sugeridos por Guerraoui e Raynal [20].

Um outro aspecto importante a considerar é que a detecção segue um padrão assíncrono. Como as suspeitas são feitas com base no padrão de troca de mensagens do algoritmo, conjectura-se ser impossível realizar a detecção de falhas por omissão se tal padrão é do tipo $1 \rightarrow n$. Ou seja, se, em algum momento da execução do algoritmo, requer-se que apenas um processo envie mensagens. Isto porque não haveria como diferenciar uma falha por omissão de um retardo na recepção da mensagem proveniente de tal processo, dado que o sistema subjacente é assíncrono [4]. Portanto, identificamos a seguinte conjectura, que se correta deriva o seguinte corolário:

Conjectura 1 *Num sistema assíncrono, é impossível detectar falhas bizantinas por omissão, de forma assíncrona, caso o padrão de troca de mensagens do algoritmo $\mathcal{A}$ seja do tipo $1 \rightarrow n$; isto é, se em determinado momento, o algoritmo $\mathcal{A}$ determina que apenas um processo envia mensagens aos demais (n).*

Corolário 1 *A forma assíncrona da detecção de falhas bizantinas só pode ser adotada por protocolos simétricos,*

nos quais todos os nós executam o mesmo papel.

4.2. Geração de suspeitas e equívocos

Geração de suspeitas. Cada suspeita (*suspicion*) sobre um processo p_i é associada a uma mensagem m requerida por $\mathcal{A}$. Requer-se, portanto, que as mensagens recebam identificadores únicos. As suspeitas são propagadas na rede e um processo correto adotará uma suspeita não gerada por ele próprio se e somente se receber pelo menos $f + 1$ ocorrências devidamente autenticadas provenientes de processos distintos. O requisito de pelo menos $f + 1$ ocorrências impede que processos maliciosos imponham suspeitas sobre processos corretos. A Figura 2 ilustra esse mecanismo para determinada porção da rede, supondo $f = 1$. Os vizinhos de um processo $p1$, faltoso por omissão, perceberão que $p1$ não está enviando as mensagens que deveria. Numa rede com f-cobertura bizantina, pelos menos dois $(f + 1)$ dos processos vizinhos a $p1$, neste caso $p2$ e $p3$, serão corretos e divulgarão uma suspeita de falha S para seus respectivos vizinhos. Qualquer processo correto da rede, por exemplo, $p10$, terá pelo menos um caminho para $p2$ ($p10p9 \cdots p5p2$) e $p3$ ($p10p3$) formado apenas por processos corretos (vide Observação 1), recebendo pelo menos duas $(f + 1)$ ocorrências de S, adotando portanto a suspeita.

Geração de equívocos. Seja p_i um processo suspeito de não ter enviado uma mensagem m. Se em algum momento um processo correto p_j receber m de p_i, p_j declarará um equívoco (*mistake*) sobre a suspeita e difundirá a mensagem m aos demais, a fim de que façam o mesmo. Na Figura 3, um processo lento $p1$, sobre o qual foi levantada uma suspeita, envia a mensagem requerida m (representada pelo envelope) a um processo correto $p2$. Numa rede com f-cobertura bizantina, haverá pelo menos um caminho formado apenas por processos corretos de cada processo correto até $p2$. Por exemplo, para $p10$, temos o caminho $p2p5 \cdots p9p10$. Logo $p10$ (e qualquer outro processo correto) receberá m e poderá retirar sua suspeita correspondente.

Esse procedimento permite que um processo malicioso, de forma intermitente, provoque uma suspeita e em seguida a revogue, levando-se a um mascaramento de parte das falhas por omissão e a uma degradação do desempenho do detector de falhas. Não é possível, no entanto, distinguir um processo com este comportamento de um processo lento ou de um período de instabilidade no canal de comunicação.

4.3. Detecção de falhas de segurança

Para ser possível a detecção das falhas de segurança, um formato de mensagens deve ser estabelecido. Cada mensagem deve também incluir um certificado que possibilite aos demais processos verificar a coerência da mesma com o algoritmo $\mathcal{A}$. Caso um processo correto detecte a invalidade de uma mensagem recebida, seja por não atender ao formato ou por não estar devidamente justificada, suspeitará permanentemente do processo remetente e encaminhará a mensagem original aos demais processos, de forma que a suspeita seja propagada. A Figura 4 representa essa situação: um processo faltoso $p1$ comete uma falha de segurança divulgando uma mensagem corrompida m (representada pelo asterisco). A falha é detectada por um vizinho correto $p2$, que encaminha a mensagem corrompida para o restante da rede. Uma vez que para cada processo correto há um caminho formado apenas por processos corretos até $p2$ (para $p10$, o caminho $p10p9 \cdots p5p2$), todos suspeitarão permanentemente de $p1$.

No protocolo, suspeitas, equívocos e provas de falhas de segurança são todos encaminhados através de uma única mensagem do tipo SUSPICION, as quais não precisam ser certificadas. Mensagens que não estejam devidamente autenticadas são descartadas, uma vez que configuram uma falha não-diagnosticável (ver Seção 3). O detector, portanto, não comete erros na detecção das falhas de segurança do algoritmo $\mathcal{A}$.

No protocolo de Kihlstrom *et al.* [12], adicionalmente, é necessário detectar quando um processo encaminha duas versões diferentes da mesma mensagem, ou seja, *mensagens mutantes*. Para isto requer-se, de um lado, que os processos corretos reencaminhem para todos os demais cada mensagem recebida, e, por outro lado, seja gerado um histórico das mensagens provenientes de cada processo. Supõe-se, no entanto, que a comunicação entre os processos é ponto a ponto. No modelo aqui proposto, em que os processos se comunicam apenas por difusão local em canais confiáveis, é natural supor que uma mensagem transmitida será recebida, com conteúdo idêntico, por todos os processos corretos, de forma que é impossível a ocorrência de mensagens mutantes. Chega-se, portanto, à seguinte observação:

Observação 2 *Em um ambiente de falhas bizantinas, a suposição de comunicação por difusão (*broadcast*) em canais confiáveis simplifica o tratamento das falhas de segurança, uma vez que os processos vizinhos do remetente têm uma visão consistente das mensagens enviadas pelo mesmo.*

5. Propriedades comportamentais do sistema

Para que o protocolo proposto neste trabalho detecte as falhas de maneira assíncrona e satisfaça as proprieda-

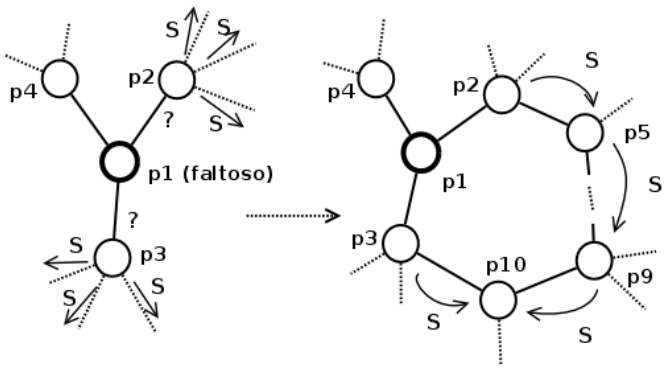

Figura 2. Geração de suspeitas no protocolo proposto ($f = 1$)

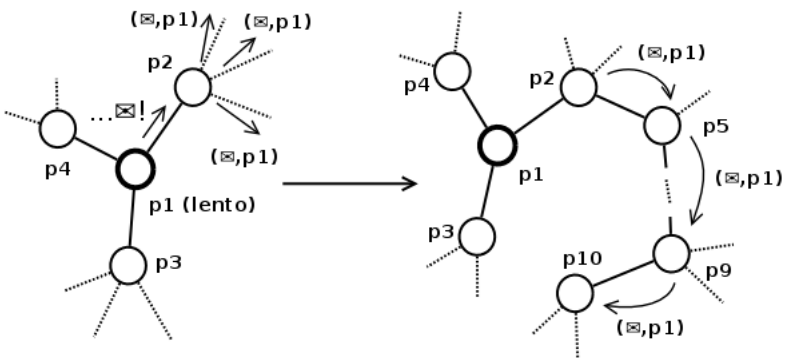

Figura 3. Geração de equívocos no protocolo proposto ($f = 1$)

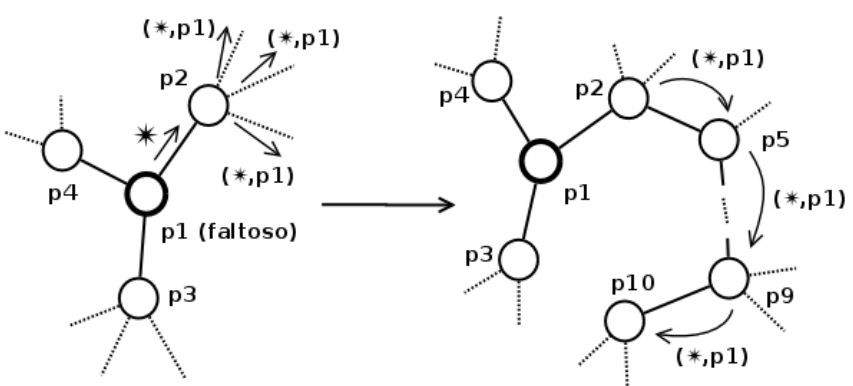

Figura 4. Detecção de falhas de segurança no protocolo proposto ($f = 1$)

des definidas para a classe $\Diamond\mathcal{S}(Byz, \mathcal{A})$, é necessário que o sistema atenda a certas propriedades comportamentais.

Propriedade 1 (Inclusão Bizantina ($Byz\mathcal{MP}$)) *Seja t um instante de tempo e KB_i^t o conjunto de processos que receberam uma mensagem* SUSPICION *de p_i até o instante t. Um processo p_i satisfaz a propriedade de inclusão bizantina $Byz\mathcal{MP}$ se:*

$Byz\mathcal{MP}(p_i) := \exists t \geq 0 : |KB_i^t| \geq 2f + 1$

Esta propriedade garante que um processo novo p_i no sistema em algum momento no futuro será conhecido por pelo menos $2f+1$ processos, dos quais pelo menos $f+1$ serão corretos. Para tanto, é necessário que p_i se comunique com pelo menos $2f + 1$ processos dentro de sua área de cobertura, enviando-lhes uma mensagem SUSPICION por difusão. Dessa forma, se p_i falhar, em algum momento no futuro pelo menos $f+1$ processos corretos suspeitarão de p_i e transmitirão a suspeita para o restante do sistema, de forma que a propriedade de completude forte bizantina do detector $\Diamond\mathcal{S}(Byz, \mathcal{A})$ seja atendida. Note-se que essas comunicações devem ser realizadas antes do início do próximo passo do algoritmo $\mathcal{A}$, de forma a garantir a propriedade de completude forte bizantina do detector de falhas; antes disso, p_i não deve ser considerado participante da computação. Se um processo novo não se comunicar com nenhum outro, é impossível atender à propriedade de completude fraca [21].

Para que a propriedade de precisão fraca após um tempo da classe $\Diamond\mathcal{S}(Byz, \mathcal{A})$ seja atendida, é necessário que exista um processo correto p_i cujas mensagens a partir de algum momento estejam sempre entre as $d - f$ primeiras recebidas pelos seus vizinhos, a cada requisição de $\mathcal{A}$. Assim sendo, após um tempo p_i não será mais suspeito por nenhum processo correto e terá quaisquer suspeitas anteriores revogadas através de mensagens de equívocos. Logo, a rede deve possuir um nó correto que atenda à propriedade de *receptividade bizantina*, definida como:

Propriedade 2 (Receptividade Bizantina ($Byz\mathcal{RP}$)) *Sejam t e u instantes de tempo e seja $rec_from_j^t$ o conjunto de pelo menos $d - f$ processos dos quais p_j recebeu a mensagem requerida por $\mathcal{A}$ no último passo em execução até o instante t. A propriedade $Byz\mathcal{RP}$ do processo correto p_i é definida como:*

$Byz\mathcal{RP}(p_i) := \exists u : \forall t > u, \forall p_j \in range_i, p_i \in rec_from_j^t$

6. Protocolo de detecção assíncrona de falhas bizantinas em sistemas dinâmicos

Os Algoritmos 1 e 2 implementam um detector de falhas da classe $\Diamond\mathcal{S}(Byz, \mathcal{A})$, desde que a rede atenda às propriedades de f-cobertura bizantina, inclusão bizantina e receptividade bizantina descritas anteriormente e que o protocolo executado pelo algoritmo $\mathcal{A}$ seja simétrico. Cada processo executa três tarefas em paralelo, descritas a seguir. Posteriormente são descritas as variáveis, primitivas e procedimentos para cada processo p_i.

T1. Geração de novas suspeitas (linhas 5-14, Algoritmo 1). Quando o algoritmo $\mathcal{A}$ requer que os processos troquem entre si uma mensagem m (linha 6), cada processo p_i aguarda receber m de pelo menos $d - f$ processos, cujos identificadores são armazenados no conjunto rec_from_i (linhas 7-8). Para os demais processos conhecidos por p_i, é adicionada uma suspeita interna de falha por omissão (linhas 9-11). Cada mensagem é então verificada quanto à sua formação e certificação (linhas 12-14, Algoritmo 1 e 5-16, Algoritmo 2). Mensagens incorretas levam a suspeitas de falhas de segurança (linhas 9-10, Algoritmo 2) e à atualização da saída do detector; mensagens corretas levam à geração de equívocos de possíveis suspeitas de falhas por omissão (linhas 12-13, Algoritmo 2).

T2. Recepção de mensagens de processos lentos e de mensagens SUSPICION (linhas 16-18, Algoritmo 1). Uma vez que as mensagens enviadas por um processo lento podem ser recebidas após a geração das suspeitas pelos seus vizinhos, é necessário que estes identifiquem tais eventos separadamente, o que é feito por esta tarefa. As mensagens são tratadas de forma similar à tarefa T1. O tratamento das mensagens SUSPICION será explanado adiante.

T3. Divulgação do novo estado de suspeitas e equívocos (linhas 20-23, Algoritmo 1). Tarefa executada periodicamente para enviar aos vizinhos de p_i (linha 22) a visão de p_i quanto às suspeitas (internas e externas), equívocos e provas de falhas de segurança. Os vizinhos de p_i receberão esta mensagem na tarefa T2 e tratá-la-ão da seguinte forma:

Atualização do estado interno (linhas 15 e 18-40, Algoritmo 2). Ao receber uma mensagem SUSPICION de um vizinho q (linha 19), um processo p_i atualiza seu estado interno com novas informações. Suspeitas internas e externas de q são adicionadas ao conjunto de suspeitas externas de p_i (linhas 20-31), possivelmente gerando novas suspeitas internas (linhas 32-34, Algoritmo 1). Note-se que será gerada uma suspeita de falha de segurança do processo q (linhas 9-10) caso a mensagem SUSPICION m esteja malformada ou não justificada. Provas de falhas de segurança e informações de equívoco são tratadas de forma similar às mensagens recebidas diretamente do remetente (linhas 34 e 38). Algumas otimizações

Algoritmo 1 Detector Assíncrono de Falhas Bizantinas em Sistemas Dinâmicos

1: **init:**
2: $output_i \leftarrow known_i \leftarrow \emptyset$; $extern_susp_i \leftarrow [][]$
3: $intern_susp_i \leftarrow mistake_i \leftarrow []$; $byzantine_i \leftarrow \emptyset$
4:
5: **Task** T1: */* geração de novas suspeitas */*
6: **when** p_i requer uma mensagem m **do**
7: **wait until** receber m devidamente autenticada pela primeira vez de pelo menos $(d - f)$ processos distintos
8: $rec_from_i \leftarrow \{p_j \mid p_i$ recebeu uma mensagem de p_j na linha 7$\}$
9: **for all** $p_j \in (known_i \setminus rec_from_i)$ **do**
10: AddInternalSusp(p_j, m)
11: **end for**
12: **for all** m_j recebida na linha 7 **from** p_j **do**
13: ValidateReceived(p_j, m_j)
14: **end for**
15:
16: **Task** T2: */* recepção de mensagens de processos lentos e do estado interno de outro processo */*
17: **upon receipt of** m devidamente autenticada **from** p_j **do**
18: ValidateReceived(p_j, m)
19:
20: **Task** T3: */* difusão do estado das suspeitas */*
21: **loop**
22: broadcast $\langle$SUSPICION, $byzantine_i$, $mistake_i$, $intern_susp_i$, $extern_susp_i\rangle$
23: **end loop**
24:
25: /* PROCEDIMENTOS AUXILIARES */
26: **procedure** AddInternalSusp(q, m):
27: $intern_susp_i[q] \leftarrow intern_susp_i[q] \cup \{m.\text{id}\}$
28: $output_i \leftarrow output_i \cup \{q\}$
29:
30: **procedure** AddExternalSusp(q, idm, p_s):
31: $extern_susp_i[q][idm] \leftarrow extern_susp_i[q][idm] \cup \{p_s\}$
32: **if** $|extern_susp_i[q][idm]| \geq f + 1$ **then**
33: AddInternalSusp(q, message(idm))
34: **end if**
35:
36: **procedure** AddMistake(q, m):
37: $mistake_i[q] \leftarrow mistake_i[q] \cup \{m\}$
38: $extern_susp_i[q][m.\text{id}] \leftarrow \emptyset$
39: $intern_susp_i[q] \leftarrow intern_susp_i[q] \setminus \{m.\text{id}\}$
40: **if** $intern_susp_i[q] = \emptyset$ **and** $\nexists\langle q, -\rangle \in byzantine_i$ **then**
41: $output_i \leftarrow output_i \setminus \{q\}$
42: **end if**

Algoritmo 2 Detector Assíncrono de Falhas Bizantinas em Sistemas Dinâmicos (continuação)

1: **procedure** AddByzantine(q, m):
2: $output_i \leftarrow output_i \cup \{q\}$;
3: $byzantine_i \leftarrow byzantine_i \cup \{\langle q, m\rangle\}$
4:
5: **procedure** ValidateReceived(q, m):
6: **if** m foi enviada diretamente por q **then**
7: $known_i \leftarrow known_i \cup \{q\}$
8: **end if**
9: **if** m não está devidamente formada **or** m não está devidamente justificada **then**
10: AddByzantine(q, m)
11: **else**
12: **if** $m.\text{id} \in intern_susp_i[q]$ **or** m foi reencaminhada **then**
13: AddMistake(q, m)
14: **end if**
15: UpdateSuspicions(q, m)
16: **end if**
17:
18: **procedure** UpdateSuspicions(q, m):
19: **if** $m = \langle$SUSPICION, $byzantine_q$, $mistake_q$, $intern_susp_q$, $extern_susp_q\rangle$ **then**
20: **for all** $p_x \in$ keys($extern_susp_q$) **do**
21: **for all** $idm_x \in$ keys($extern_susp_q[p_x]$) devidamente autenticado $\mid idm_x \notin$ ids($mistake_i[p_x]$) **do**
22: **for all** $p_y \in extern_susp_q[p_x][idm_x]$ **do**
23: AddExternalSusp(p_x, idm_x, p_y)
24: **end for**
25: **end for**
26: **end for**
27: **for all** $p_x \in$ keys($intern_susp_q$) **do**
28: **for all** $idm_x \in intern_susp_q[p_x]$ devidamente autenticado $\mid idm_x \notin$ ids($mistake_i[p_x]$) **do**
29: AddExternalSusp(p_x, idm_x, q)
30: **end for**
31: **end for**
32: **for all** $p_x \in$ keys($mistake_q$) **do**
33: **for all** $m_x \in mistake_q[p_x]$ devidamente autenticada **do**
34: ValidateReceived(p_x, m_x)
35: **end for**
36: **end for**
37: **for all** $\langle p_x, m_x\rangle \in byzantine_q \mid m_x$ está devidamente autenticada **do**
38: ValidateReceived(p_x, m_x)
39: **end for**
40: **end if**

nesse procedimento foram removidas nesta versão do protocolo, a fim de simplificar as demonstrações da Seção 7. O leitor interessado poderá encontrá-las em [22].

VARIÁVEIS:
• $output_i$: armazena a saída do detector de falhas, isto é, o conjunto de identificadores dos processos de que p_i suspeita terem falhado;
• $known_i$: armazena o conjunto dos processos que se comunicaram com p_i, isto é, sua suposta vizinhança. É atualizada a cada recepção de mensagem m, seja m do tipo SUSPICION ou uma mensagem inerente a $\mathcal{A}$;
• $extern_susp_i$: matriz que armazena suspeitas externas (geradas por outros processos). A matriz é indexada por um identificador de processo q e por um identificador de mensagem idm. Cada entrada armazena o conjunto de processos dos quais p_i recebeu suspeitas de q não ter enviado a mensagem com identificador idm;
• $intern_susp_i$: vetor de suspeitas internas. Uma suspeita interna é gerada pela não-recepção de uma mensagem requerida por $\mathcal{A}$ ou pela existência de pelo menos $f + 1$ suspeitas externas sobre um mesmo par processo-mensagem;
• $mistake_i$: vetor que armazena, para cada processo aplicável p_j, o conjunto de equívocos relacionados a p_j, na forma de mensagens requeridas por $\mathcal{A}$ sobre as quais foram levantadas suspeitas;
• $byzantine_i$: conjunto de duplas do tipo $\langle$processo, mensagem$\rangle$ que provam comportamento bizantino do processo associado. A notação $\langle p, -\rangle$ indica "qualquer dupla associada ao processo p";
• rec_from_i: conjunto de processos dos quais p_i recebeu a mensagem requisitada por $\mathcal{A}$.

PRIMITIVAS:
• m.id - retorna o identificador da mensagem m;
• message(idm) - retorna a mensagem associada ao identificador idm;
• verificação de boa formação, justificação (certificação do conteúdo) e autenticação das mensagens;
• requisição por $\mathcal{A}$ de uma determinada mensagem;
• broadcast m - difunde a mensagem m para os vizinhos de p_i;
• keys(v) - retorna o conjunto de índices de um vetor dinâmico v;
• ids(s) - retorna o conjunto de identificadores associados às mensagens do conjunto s.

PROCEDIMENTOS AUXILIARES:
• AddInternalSusp(q, m) (linhas 26-28, Algoritmo 1): adiciona uma suspeita interna sobre o processo q em relação à mensagem m;
• AddExternalSusp(q, idm, p_s) (linhas 30-34, Algoritmo 1): adiciona uma suspeita externa sobre o processo q proveniente do processo p_s em relação à mensagem com identificado idm. Adicionalmente, caso existam $f + 1$ ou mais suspeitas externas sobre q em relação a message(idm), gera uma suspeita interna equivalente, caso ainda não exista;
• AddMistake(q, m) (linhas 36-42, Algoritmo 1): adiciona um equívoco sobre a suspeita de q não ter enviado m, retirando todas as suspeitas internas e externas associadas. Caso q não tenha outras suspeitas nem tenha apresentado comportamento bizantino, remove-o da saída do detector de falhas;
• AddByzantine(q, m) (linhas 1-3, Algoritmo 2): adiciona q permanentemente à lista de processos maliciosos (e, conseqüentemente, à saída do FD), juntamente com a mensagem m associada à falha bizantina, servindo como prova da falha;
• ValidateReceived(q, m) (Algoritmo 2, linhas 5-16): verifica a validade (boa formação e justificação) da mensagem m recebida de q, retirando quaisquer suspeitas associadas ao par (q, m), caso m seja válida, e gerando uma suspeita de falha de segurança, caso contrário. Adicionalmente, atualiza o conjunto de nós conhecidos por p_i ($known_i$) e repassa as mensagens ao procedimento a seguir, de forma a atualizar o estado de suspeitas;
• UpdateSuspicions(q, m) (Algoritmo 2, linhas 18-40): caso m seja do tipo SUSPICION, atualiza o estado interno de p_i com as informações de falhas contidas em m.

7. PROVA DE CORREÇÃO

A correção do algoritmo é definida em função das propriedades de completude forte bizantina e precisão fraca após um tempo da classe $\Diamond\mathcal{S}(Byz, \mathcal{A})$ de detectores de falhas. Dessa forma, os argumentos são exibidos para cada propriedade separadamente.

7.1. COMPLETUDE FORTE BIZANTINA

Lema 1 *Se um processo p_i nunca enviar uma mensagem m, jamais algum processo correto efetuará uma chamada ao procedimento* AddMistake *com parâmetros* (p_i, m).

Prova: Suponha-se, por contradição, que algum processo correto p_j efetue uma chamada a AddMistake com parâmetros (p_i, m). Note-se que a única chamada ao procedimento AddMistake está no procedimento ValidateReceived, na linha 13 do Algoritmo 2. O procedimento ValidateReceived, por sua vez, é chamado nos seguintes casos: (1) na recepção de mensagens requeridas por $\mathcal{A}$, seja durante o mecanismo de espera da tarefa T1, seja posteriormente na tarefa T2 (linhas 13 e 18 do Algoritmo 1, respectivamente); (2) na recepção de mensagens SUSPICION, na tarefa T2; (3) na atualização do estado interno com informações de vizinhos (procedimento UpdateSuspicions, linhas 34 e 38 do Algoritmo 2). Em todos os

casos, a autenticação de m é verificada (respectivamente, nas linhas 7 e 17 do Algoritmo 1 e nas linhas 33 e 37 do Algoritmo 2). Desse fato e do uso de canais confiáveis, um processo bizantino não pode enviar m em lugar de p_i. O caso (2), especificamente, não poderia causar uma chamada a AddMistake, já que não existem suspeitas relacionadas a mensagens SUSPICION e mensagens SUSPICION corretas não são reencaminhadas.

Em todos os casos, chega-se à contradição de ser necessário que p_i envie m em algum momento, do que o lema segue. □

Lema 2 *Seja p_i um processo faltoso por omissão. Em algum momento no futuro, todo processo correto p_j em Π incluirá p_i permanentemente em $output_j$.*

Prova: Seja m a primeira mensagem requerida por $\mathcal{A}$ não enviada por p_i. Seja t o instante em que $\mathcal{A}$ requer m de p_i e u o primeiro instante tal que $|KB_i^u| \geq 2f+1$, sendo KB_i^u como definido na Propriedade 1; um tal instante u existe devido a $Byz\mathcal{MP}$. Tem-se que $t \geq u$, pois antes de u p_i não participava da computação distribuída (ver Seção 5).

Se p_j está em KB_i^t, então p_j recebeu uma mensagem do tipo SUSPICION de p_i até o instante t, e portanto p_i está em $known_j$, conforme as linhas 17-18 do Algoritmo 1 e 6-8 do Algoritmo 2. Quando $\mathcal{A}$ requerer m a p_j, este aguardará somente até receber m de $d-f$ processos distintos (linhas 6-7 do Algoritmo 1); esse evento acontecerá em algum momento, já que no máximo f processos são faltosos e $|range_j| \geq d$. Como p_i não envia m, não será incluído em rec_from_j (linha 8 do Algoritmo 1). Conforme as linhas 9-11 e 27-28 do Algoritmo 1, m.id será incluído em $intern_susp_j[p_i]$ e p_i em $output_j$. Uma vez que p_i também não enviará m em outro momento posterior, pelo Lema 1 e pelas linhas 39-42 do Algoritmo 1, m.id nunca será removido de $intern_susp_j[p_i]$ nem p_i de $output_j$.

Se p_j não está em KB_i^t, no entanto, uma vez que a rede possui f-cobertura bizantina, existe ao menos um caminho P formado apenas por processos corretos entre p_j e cada p_k correto em KB_i^t; se houver mais de um tal caminho, tome-se o de latência total mínima. Prova, por indução no comprimento de P, de que em algum momento p_k é adicionado a $extern_susp_j[p_i][m.\text{id}]$: se P tem comprimento 1, p_j é vizinho de p_k; nesse caso, em algum momento p_k difunde (linha 22 do Algoritmo 1) uma mensagem SUSPICION s com a informação autenticada de que $m.\text{id} \in intern_susp_k[p_i]$; como os canais são confiáveis, em algum momento p_j recebe s na linha 17 do Algoritmo 1. Como p_k é correto, s estará devidamente autenticada, formada e justificada; pelo Lema 1, $m \notin mistake_j[p_i]$; assim, pela linha 18 do Algoritmo 1 e pelas linhas 15 e 27-31 do Algoritmo 2, p_k é adicionado a $extern_susp_j[p_i][m.\text{id}]$ na linha 31 do Algoritmo 1 e a afirmação vale. Se o comprimento de P é maior que 1, pode-se supor por hipótese de indução que a afirmação vale para o caminho $P - p_j$ entre p_k e um processo correto p_l, visto que tal caminho possui comprimento menor que P. Notadamente p_l é vizinho de p_j em P; por hipótese de indução, em algum momento p_k é adicionado a $extern_susp_l[p_i][m.\text{id}]$ e, futuramente, p_l difunde (linha 22 do Algoritmo 1) uma mensagem SUSPICION s com essa informação autenticada por p_k; como os canais são confiáveis, em algum momento p_j recebe s na linha 17 do Algoritmo 1. Como p_l é correto, s estará devidamente autenticada, formada e justificada; pelo Lema 1, $m \notin mistake_j[p_i]$; assim, pela linha 18 do Algoritmo 1 e pelas linhas 15 e 20-26 do Algoritmo 2, p_k é adicionado a $extern_susp_j[p_i][m.\text{id}]$ na linha 31 do Algoritmo 1 e a afirmação vale.

Desse fato, de $|KB_i^t| \geq 2f+1$ e de existirem no máximo f processos faltosos segue que em algum momento p_j executa a linha 33 do Algoritmo 1 e, pelas linhas 27-28, adiciona m.id a $intern_susp_j[p_i]$ e p_i a $output_j$. Novamente, segue do Lema 1 que p_i nunca será removido de $output_j$. □

Lema 3 *Seja p_i um processo faltoso por comissão (isto é, que cometa uma falha de segurança detectável). Em algum momento no futuro, todo processo correto p_j em Π, adiciona permanentemente p_i a $output_j$.*

Prova: Uma falha por comissão consiste no envio por p_i de uma mensagem m em desacordo com $\mathcal{A}$, isto é, uma mensagem mal formada ou não justificada (ver Seção 3); o uso de comunicação por difusão impede a ocorrência de mensagens mutantes (ver Seção 4). Além disso, m é uma mensagem autenticada; caso contrário, configura-se uma falha não-diagnosticável (ver Seção 3).

Devido à propriedade de f-cobertura bizantina, existe um caminho P, entre p_i e cada processo correto p_j, formado apenas por processos corretos, exceto por p_i; se houver mais de um tal caminho, tome-se o de latência total mínima. Prova, por indução no comprimento de P, de que em algum momento p_j adiciona $\langle p_i, m\rangle$ a $byzantine_j$ e p_i a $output_j$: se o comprimento de P é 1, então $p_j \in range_i$ e, já que os canais são confiáveis e m é autenticada, p_j recebe m em algum momento na linha 7 ou na linha 17 do Algoritmo 1. Em ambos os casos, é feita uma chamada ao procedimento ValidateReceived (linhas 13 e 18 do Algoritmo 1), que verifica na linha 9 do Algoritmo 2 a invalidade de m. O procedimento AddByzantine (linhas 1-3 do Algoritmo 2), por sua vez, adiciona p_i a $output_j$ e $\langle p_i, m\rangle$ a $byzantine_j$, do que a afirmação segue. Se o comprimento de P é maior que 1, a afirmação é válida por hipótese de indução para o caminho $P - p_j$

entre p_i e algum processo correto p_k vizinho de p_j. Nesse caso, $\langle p_i, m\rangle$ está em $byzantine_k$ e, em algum momento posterior, p_k difunde (linha 22 do Algoritmo 1) uma mensagem SUSPICION s com essa informação; como os canais são confiáveis, em algum momento p_j recebe s na linha 17 do Algoritmo 1. Como p_l é correto, s estará devidamente autenticada, formada e justificada; assim, por m ser autenticada, pela linha 18 do Algoritmo 1 e pelas linhas 15 e 37-39 do Algoritmo 2, p_j efetua uma chamada a ValidateReceived que confirma a invalidade de m; portanto p_i é adicionado a $output_j$ e $\langle p_i, m\rangle$ a $byzantine_j$ e a afirmação vale.

Desse fato e uma vez que p_j só remove p_i de $output_j$ se não houver um par $\langle p_i, -\rangle$ em $byzantine_j$ (linhas 40-42 do Algoritmo 1), p_i é adicionado definitivamente a $output_j$ e o lema segue. □

7.2. Precisão fraca após um tempo

Lema 4 *Se p_i e p_j são processos corretos então, durante toda a computação, p_j jamais efetua uma chamada ao procedimento* AddByzantine *tendo p_i como primeiro parâmetro.*

Prova: Note que a única chamada a AddByzantine está na linha 10 do Algoritmo 2, no procedimento ValidateReceived. Conforme a linha 9, essa chamada só é efetuada por um processo correto se p_i enviar uma mensagem mal formada ou não justificada, o que não é possível, uma vez que p_i é correto. Se um processo bizantino enviar uma tal mensagem no lugar de p_i, p_j a descartará, uma vez que os canais de comunicação são confiáveis e p_j verifica a autenticação de toda mensagem que recebe (linhas 7 e 17 do Algoritmo 1 e linhas 33 e 37 do Algoritmo 2), logo o lema segue. □

Lema 5 *Seja p_i um processo correto e m uma mensagem requerida por $\mathcal{A}$. Se todo nó em $range_i$ receber m de p_i na linha 7 do Algoritmo 1, jamais algum processo correto p_j em Π efetuará uma chamada a* AddInternalSusp *com parâmetros (p_i, m).*

Prova: O procedimento AddInternalSusp é chamado em duas situações: (1) na tarefa T1, durante a recepção de mensagens requeridas por $\mathcal{A}$ e (2) no procedimento AddExternalSusp, ao identificar a ocorrência de $f + 1$ ou mais suspeitas externas de p_i não ter enviado m. Pela hipótese do lema, processos em $range_i$ adicionam p_i a rec_from_j na linha 8 do Algoritmo 1, logo não chamam AddInternalSusp na linha 10 com parâmetros (p_i, m). Um processo p_j fora de $range_i$ não pode receber mensagens diretamente de p_i, logo não adiciona p_i a $known_j$ em nenhum momento (linhas 6-8 do Algoritmo 2) nem efetua a chamada a AddInternalSusp da linha 10 com parâmetros (p_i, m). Em ambas as situações, o caso (1) confirma o lema.

Quanto ao caso (2), note-se que chamadas a AddExternalSusp só são realizadas nas linhas 23 e 29 do Algoritmo 2. Uma vez que um processo correto só atualiza seu conjunto $extern_susp$ no procedimento AddExternalSusp, toda suspeita externa proveniente de um processo correto foi gerada como uma suspeita interna (ver linhas 22 e 28 do Algoritmo 1). Pelo argumento do caso (1), um processo correto p_j nunca adiciona m.id a $intern_susp_j[p_i]$ através tarefa T1; se um processo bizantino p_k adicionar p_j a $extern_susp_k[p_i][m.\text{id}]$, um processo correto não adotará tal suspeita, uma vez que sua autenticação é verificada na linha 21 do Algoritmo 2. Um processo faltoso p_j pode, no entanto, adicionar m.id a $intern_susp_j[p_i]$ e autenticar essa informação, mas só existem no máximo f processos faltosos, logo nenhum processo correto chama AddInternalSusp na linha 33 do Algoritmo 1 com parâmetros (p_i, m).

Uma vez que todos os casos possíveis foram analisados, o lema segue. □

Lema 6 *Seja p_i um processo correto. Se existe um mensagem m e um processo correto p_j tal que $m.\text{id} \in intern_susp_j[p_i]$ em algum instante da computação, então em algum momento no futuro p_j efetua uma chamada a* AddMistake *com parâmetros (p_i, m).*

Prova: Se p_j está em $range_i$, como p_i é correto, em algum momento p_j recebe m de p_i (devidamente autenticada, formada e justificada) na linha 17 do Algoritmo 1. Pela hipótese do lema, $m.\text{id} \in intern_susp_j[p_i]$, assim, pelas linhas 18 do Algoritmo 1 e pelas linhas 9 e 12 do Algoritmo 2, p_j efetua uma chamada a AddMistake com parâmetros (p_i, m) na linha 13 do Algoritmo 2.

Se p_j não está em $range_i$, por argumento semelhante ao feito no lema anterior, $p_i \notin known_j$. Então, algum outro processo correto p_k em $range_i$ gerou a suspeita, isto é, existe p_k em $range_i$ tal que $m.\text{id} \in intern_susp_k[p_i]$. Como a rede possui f-cobertura bizantina, existe um caminho P formado apenas por processos corretos entre p_k e p_j; se houver mais de um tal caminho, tome-se o de latência total mínima. Prova, por indução no comprimento de P, de que em algum momento cada processo p_l em P chama AddMistake com parâmetros (p_i, m) e portanto $m \in mistake_l[p_i]$: se o comprimento de P é zero, P contém apenas p_k e, pelo argumento do parágrafo anterior, a afirmação vale. Se o comprimento de P é maior que zero então, por hipótese de indução, a afirmação vale para o caminho $P - p_j$ entre p_k e um processo p_l e, em algum momento, $m \in mistake_l[p_i]$. Futuramente, p_l difunde uma mensagem SUSPICION s com m devidamente autenticada em $mistake_l[p_i]$; como

os canais são confiáveis, em algum momento p_j recebe s na linha 17 do Algoritmo 1. Como p_l é correto, s estará devidamente autenticada, formada e justificada e, pela linha 18 do Algoritmo 1 e pelas linhas 19 e 32-36 do Algoritmo 2, p_j chama ValidateReceived com parâmetros (p_i, m). Como p_i é correto, m está devidamente formada e justificada. Como m foi reencaminhada por p_l, p_j chama AddMistake com parâmetros (p_i, m) na linha 13 do Algoritmo 2 (ver linhas 9 e 12-14) e a afirmação vale; portanto, o lema segue. □

Lema 7 *Seja p_i um processo correto que atenda a $Byz\mathcal{RP}$. Em algum momento no futuro, todo processo correto p_j em Π é tal que $p_i \notin output_j$.*

Prova: Do Lema 4, afirma-se que p_i jamais será adicionado a $output_j$ na linha 2 do Algoritmo 2. Pela propriedade $Byz\mathcal{RP}$, existe um instante t após o qual toda mensagem m requerida por $\mathcal{A}$ é recebida pelos vizinhos de p_i na linha 7 do Algoritmo 1; do Lema 5 verifica-se que p_j não adiciona p_i a $output_j$ em uma chamada a AddInternalSusp com parâmetros (p_i, m). Para toda mensagem m' requerida por $\mathcal{A}$ antes de t, se em algum instante $m' \in intern_susp_j[p_i]$, pelo Lema 6, em algum momento futuro p_j chama AddMistake com parâmetros (p_i, m'). Assim, pela linha 39 do Algoritmo 1, em algum momento tem-se que $intern_susp_j[p_i] = \emptyset$; pelo Lema 4, não existe $\langle p_i, -\rangle$ em $byzantine_j$, logo p_i é removido de $output_j$ na linha 41 do Algoritmo 1 e o lema segue. □

Teorema 1 *Os Algoritmos 1 e 2 implementam um detector de falhas da classe $\Diamond\mathcal{S}(Byz, \mathcal{A})$.*

Prova: O teorema segue dos Lemas 2, 3 e 7 e da definição da classe $\Diamond\mathcal{S}(Byz, \mathcal{A})$ (ver Seção 3). □

8. Conclusão e trabalhos futuros

Este trabalho apresentou um detector de falhas bizantinas com duas características inovadoras: (i) ele foi projetado para sistemas distribuídos dinâmicos, cujo conjunto de participantes na rede é desconhecido e, além disso, (ii) tem a característica de ser assíncrono, isto é, não se utiliza de temporizadores para a detecção das falhas de progresso. Para que a detecção seja realizada de forma assíncrona, conjectura-se ser necessário que o algoritmo que utiliza o detector de falhas seja simétrico, isto é, que a cada passo todos os processos troquem mensagens entre si. Um aspecto interessante é que a comunicação por difusão (*broadcast*), típica das redes sem fio, simplifica o tratamento de falhas bizantinas, uma vez que os processos vizinhos de um remetente têm uma visão uniforme das mensagens por ele enviadas. Especificamente, não é necessário tratar a ocorrência de *mensagens mutantes*, isto é, quando um mesmo processo envia a mesma mensagem com conteúdos diferentes para diferentes processos.

Objetivam-se como trabalhos futuros (i) estender o protocolo para prover tolerância à mobilidade dos nós; (ii) implementar o protocolo, com o objetivo de avaliar seu desempenho e viabilidade prática; (iii) provar ou contra-exemplificar a impossibilidade de realização de detecção de falhas de forma assíncrona em sistemas sujeitos a falhas bizantinas com comunicação do tipo $1 \rightarrow n$.

Referências

[1] A. Mostefaoui, M. Raynal, C. Travers, S. Patterson, D. Agrawal, and A. El Abbadi. From Static Distributed Systems to Dynamic Systems. In *Proc. of the 24th IEEE Symp. on Reliable Distributed Systems*, pages 109–118, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

[2] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[3] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance and Proactive Recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, 2002.

[4] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

[5] M. Correia, P. Veríssimo, and N. F. Neves. Byzantine Consensus in Asynchronous Message-Passing Systems: a Survey. In *Resilience-building Technologies: State of Knowledge*, Deliverable D12, Part Algo, chapter 1. RESIST Network of Excellence, September 2006.

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

[7] M. Larrea, A. Fernández, and S. Arévalo. Optimal implementation of the weakest failure detector for solving consensus. In *Proc. of the 19th IEEE Symp. on Reliable Distributed Systems*, pages 52–59, Washington, DC, USA, 2000. IEEE Computer Society.

[8] I. Gupta, T. D. Chandra, and G. S. Goldszmidt. On scalable and efficient distributed failure detectors.

In *Proc. of the 20th Annual ACM Symp. on Principles of Distributed Computing*, pages 170–179, New York, NY, USA, 2001. ACM Press.

[9] R. Friedman and G. Tcharny. Evaluating failure detection in mobile ad-hoc networks. *Int. Journal of Wireless and Mobile Computing*, 1(8), 2005.

[10] A. Mostefaoui, E. Mourgaya, and M. Raynal. Asynchronous Implementation of Failure Detectors. In *Proc. of the 2003 Int. Conf. on Dependable Systems and Networks*, page 351, Los Alamitos, CA, USA, 2003. IEEE Computer Society.

[11] P. Sens, F. Greve, L. Arantes, M. Bouillaguet, and V. Simon. Um Detector de Falhas Assíncrono para Redes Móveis e Auto-Organizáveis. In *Anais do Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*, Rio de Janeiro, RJ, Brazil, 2008. Sociedade Brasileira de Computação.

[12] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. Byzantine Fault Detectors for Solving Consensus. *The Computer Journal*, 46(1):16–35, 2003.

[13] R. Baldoni, J.-M. Hélary, and S. T. Piergiovanni. A Component-Based Methodology to Design Arbitrary Failure Detectors for Distributed Protocols. In *Proc. of the 10th IEEE Int. Symp. on Object and Component-Oriented Real-Time Distributed Computing*, pages 51–61, Washington, DC, USA, 2007. IEEE Computer Society.

[14] B. Awerbuch, D. Holmer, C. Nita-Rotaru, and H. Rubens. An on-demand secure routing protocol resilient to byzantine failures. In *Proc. of the 1st ACM Workshop on Wireless Security*, pages 21–30, New York, NY, USA, 2002. ACM.

[15] M. K. Aguilera, W. Chen, and S. Toueg. Heartbeat: A Timeout-Free Failure Detector for Quiescent Reliable Communication. In *Proc. of the 11th Int. Workshop on Distributed Algorithms*, pages 126–140, London, UK, 1997. Springer-Verlag.

[16] D. Malkhi and M. Reiter. Unreliable intrusion detection in distributed computations. In *Proc. 10th Computer Security Foundations Workshop*, pages 116–124, Rockport, MA, 1997. IEEE Computer Society Press, Los Alamitos, CA.

[17] J. R. Douceur. The sybil attack. In *Revised Papers from the First Int. Workshop on Peer-to-Peer Systems*, pages 251–260, London, UK, 2002. Springer-Verlag.

[18] K. Tang and M. Gerla. MAC reliable broadcast in ad hoc networks. In *IEEE Military Communications Conference, 2001. MILCOM 2001. Communications for Network-Centric Operations: Creating the Information Force*, volume 2, pages 1008–1013, 2001.

[19] M.-T. Sun, L. Huang, A. Arora, and T.-H. Lai. Reliable MAC Layer Multicast in IEEE 802.11 Wireless Networks. In *ICPP '02: Proceedings of the 2002 International Conference on Parallel Processing*, pages 527–536, Washington, DC, USA, 2002. IEEE Computer Society.

[20] R. Guerraoui and M. Raynal. The Information Structure of Indulgent Consensus. *IEEE Trans. Comput.*, 53(4):453–466, 2004.

[21] A. Fernández, E. Jiménez, and S. Arévalo. Minimal system conditions to implement unreliable failure detectors. In *12th Pacific Rim Int. Symp. on Dependable Computing*, pages 63–72, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[22] M. S. de Lima and F. G. P. Greve. Protocolo Assíncrono para Detecção de Falhas Bizantinas em Sistemas Distribuídos Dinâmicos. In *Anais do Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*, pages 887–900, Recife, PE, Brazil, May 2009. Sociedade Brasileira de Computação.