Um Detector de Falhas Bizantinas Assíncrono Aplicada à Computação em Nuvem

Clésio Matos[1], Fabíola Greve[1]

[1]Departamento de Ciência da Computação – Universidade Federal da Bahia (UFBA)
Av. Adhemar de Barros, s/n – CEP 40.170-110 – Salvador – BA– Brasil

{clesiormatos,fabiola}@dcc.ufba.br

Abstract. *Modern distributed systems based on cloud computing are inherently dynamic, and the design of dependable services that can cope with the dynamics of this environment is a challenge. Byzantine failure detectors provide an elegant approach to implement fault tolerance; however, few works have been appeared. Thus, this paper aims at proposing an unreliable implementation for detecting byzantine faults in a dynamic and asynchronous environment, applying it in the cloud storage and data replication. The algorithm brings the interesting feature to propose weak time constraints of an asynchronous system to the cloud, not using time limits for fault detection.*

Resumo. *Os sistemas distribuídos modernos baseados em computação em nuvem são inerentemente dinâmicos, e a concepção de serviços confiáveis que lidem com a dinamicidade deste ambiente é um desafio. Os detectores de falhas bizantinas produzem uma abordagem elegante para a implementação segura de tolerância a falhas. Assim, este artigo tem por objetivo propor uma implementação segura para a detecção de falhas bizantinas num ambiente dinâmico e assíncrono, aplicando-o no armazenamento e replicação de dados da nuvem. O algoritmo possui a característica interessante de propor as fracas restrições de tempo de um sistema assíncrono para a nuvem, não utilizando limites temporais para detecção de falhas.*

1. Introdução

O cenário de computação distribuída está evoluindo rapidamente para integração de sistemas dinâmicos e auto-organizáveis, como é o caso da computação em nuvem. Ela propõe um novo paradigma para a gestão de infraestrutura, oferecendo possibilidades para implantar diversos recursos em ambientes distribuídos. Seu objetivo é compartilhar recursos em nuvem através de um modelo de *utility computing*, que permite o provisionamento sob demanda de recursos computacionais, operado através de máquinas virtuais em *data centers* de provedores distribuídos [Ganesh et al. 2014].

Para manter um sistema distribuído confiável, eficiente e tolerantes a falhas, faz-se necessário dispor de recursos que identifiquem e implementem mecanismos de contorno a falhas. No entanto, a concepção de serviços confiáveis que lidem com a alta dinamicidade gerada pelo fornecimento de recursos flexíveis é um desafio. A principal dificuldade encontrada para a resolução destes problemas é causada pela complexidade de se detectar as falhas existentes no sistema. Um detector de falhas (*FD*) é um serviço fundamental capaz de ajudar no desenvolvimento de sistemas distribuídos tolerantes a falhas, ele fornece periodicamente uma lista de processos suspeitos de terem falhado. Neste trabalho, apresenta-se o interesse especial nos detectores de falhas da classe não confiável, denotado por ◊S, que contém os recursos mínimos para detecção em

ambiente assíncrono. Esses *FD*s podem cometer um número arbitrário de erros, sendo que após um tempo, um processo correto nunca é suspeito e todos os processos falhos serão suspeitos permanentemente [Chandra and Toueg 1996].

A computação em nuvem se torna dinâmica na medida em que seus recursos podem ser provisionados e selecionados automaticamente em tempo de execução com base em solicitações existentes, criando redes de filiação dinâmicas ao longo do tempo. Como a arquitetura em nuvem tipicamente combina um grande número de módulos heterogêneos e dispersos geograficamente, ligados através da Internet, a confiabilidade da aplicação não depende só do sistema em si, mas também do nó que esteja implantada e da Internet imprevisível [Ganesh et al. 2014]. Portanto, os protocolos distribuídos clássicos são inadequados para esse novo contexto, uma vez que eles fazem a suposição de que todo o sistema é estático e sua composição é previamente conhecida. Além disso, a computação em nuvem está crescendo suportada por recurso e serviço *on-line*, o que torna mais provável falhas maliciosas e consequências mais graves. Por conseguinte, as soluções avançadas capazes de se adaptarem dinamicamente da aplicação real e ainda melhorar o seu desempenho, devem ser desenvolvidas [Garraghan et al. 2011].

Desta feita, este artigo tem o intuito de contribuir com um estudo sobre a detecção de falhas bizantinas para computação em nuvem aplicada ao serviço de *storage,* armazenamento e replicação de dados, através de métodos de comunicação segura. O FD é utilizado pelo serviço como um oráculo que informa um grupo de processos falhos na rede, incapazes de atender as requisições de armazenamento e replicação com segurança. Ressalta-se que o serviço de *storage* é um dos principais recursos da nuvem e está associado a um alto custo de recuperação de falhas, pois todas as réplicas devem conter a mesma informação, ordenada, obtida através de protocolos avançados como exemplo a sincronia de visões e o acordo [Greve 2005]. Assim, identificar os processos falhos e evitar sua utilização é crucial para garantir o desempenho do sistema de nuvem.

As principais contribuições deste trabalho são: *i)* criação de um novo algoritmo de detecção de falhas bizantinas para a computação em nuvem aplicado na construção de serviços confiáveis; *ii)* adoção de uma estratégia de detecção de falhas bizantinas sem a utilização de mecanismos de tempo (*time free*) e baseado em ambiente assíncrono, modelando assim, a dinamicidade dos sistemas reais, o que constitui um grande desafio em sistemas distribuídos; *iii)* utilização mecanismos de segurança de comunicação, criptografia de chave pública e privada e *hash*, possibilitando detectar falhas na presença de agentes maliciosos.

O restante deste artigo está organizado da seguintes forma: A Seção 2 discute alguns trabalhos relacionados. A Seção 3 apresenta o modelo de ambiente do detector de falhas. Na Seção 4 é apresentado o algoritmo, seguido pelas provas de correção apresentadas na Seção 5. Por fim, as conclusões e trabalhos futuros na Seção 6.

2. Trabalhos Relacionados

Grande parte das implementações de detectores de falhas são ligados a falhas benignas em ambiente síncronos [Poledna 1996]. Outro seguimento de trabalhos foca nas falhas malignas, ou bizantinas, que propõe soluções baseadas no envio de *heartbeats*, através controle do *timeout* da comunicação [Ramasamy and Cachin 2005].

Avançando neste sentido, tem-se o trabalho de Greve et al. (2012), que propõe a adoção de uma abordagem livre de tempo (não utiliza *timeout*) para efetuar a detecção de falhas bizantinas em sistemas dinâmicos. A proposta de Sivakami and Nawaz (2011) apresenta um protocolo de roteamento tolerante a falhas bizantinas, que usa um detector de falhas malignas na escolha do melhor caminho.

No contexto de falhas bizantinas, encontra-se um novo grupo de trabalhos que aplica a tolerância a falhas no ambiente de computação em nuvem. Este é o caso de Fan et. al (2012), que propõe um modelo de organização de processos na nuvem capaz de detectar falhas em componentes. Destaca-se também o trabalho de Garraghan (2011) que desenvolveu um framework para a criação de sistemas tolerantes a falhas bizantinas aplicadas à nuvens federadas, demonstrando os resultados iniciais desta abordagem.

Todas as contribuições citadas anteriormente são de grande valia para a evolução da pesquisa de falhas bizantinas, buscando soluções diversas para essa classe de problemas. Entretanto, não fazem a aplicação real da detecção de falhas em ambientes verdadeiramente assíncronos (isentos de limites temporais) com garantias de segurança de comunicação, ou ainda não são adaptáveis aos cenários dinâmicos modernos, como é o caso da computação em nuvem. Assim, o presente trabalho busca incorporar a detecção de falhas bizantinas na computação em nuvem, assumindo as fracas prerrogativas temporais de um sistema assíncrono.

3. Modelo para Detector de Falhas Bizantinas Assíncrono

3.1. Arquitetura do Sistema

Os sistemas distribuídos modernos são inerentemente dinâmicos, contendo uma população dinâmica de nós. Assim, a computação em nuvem é composta de *data centers* dispersos geograficamente que efetuam a computação e o armazenamento dos dados. Essa realidade é ainda mais evidenciada assumindo a utilização de nuvens federadas ou nuvens *Mashups* [Garraghan 2011]. Logo, quantidade e qualidade dos processos que compõem cada módulo são dinâmicas, pois a cada momento novos recursos podem ser provisionados para atender novas requisições, ou ainda, requisições antigas podem ser migradas para outros processos servidores, a fim de obter melhores resposta do sistema. Este dinamismo da computação em nuvem provê as seguintes propriedades ao modelo proposto: *i*) população dinâmica de nós, *ii*) conhecimento parcial sobre o conjunto de processos [Fan et. al 2012].

A comunicação entre todos os módulos heterogêneos que compõem o sistema em nuvem é dada pela troca de pacotes através da internet, estando sujeita a atrasos arbitrários e perdas [Ganesh et al. 2014], o que juntamente com as caraterísticas destacadas anteriormente, pode ser enxergado com um modelo de sistema assíncrono. Isto provê mais três propriedades ao modelo: *iii*) não existem suposições sobre a velocidade relativa dos processos, *iv*) não existem limites temporais para a transferência da mensagem[1], e por fim, *v*) não existe um relógio global conhecido para a coordenação dos processos dispersos. A partir desta caracterização é possível determinar o modelo de ambiente dinâmico e assíncrono, foco deste trabalho [Greve et al. 2012].

[1] Ressalta-se que o algoritmo proposto está inserido dentro da arquitetura da nuvem, e ao invés de se contrapor com as garantias de SLA (*service level agreement*) da nuvem, torna-se um suporte interno para assegurar a funcionalidade destas garantias.

Neste ambiente complexo podem ocorrer falhas em processos e na comunicação por vários motivos: porque um processo parou de funcionar (falha por parada), porque deu lugar a outro processo e esta informação ainda não está presente nos demais nós da rede, ou ainda, porque um processo executou ações não planejadas dentro do sistema (falhas bizantinas) [Fan et. al 2012]. Nesta última está concentrado o interesse deste trabalho. É razoável pensar que dentro de um *data center* as falhas malignas não ocorrem pelo nível de controle implementado nesse ambiente. Contudo, estas falhas podem ser oriundas de incidentes (ex. erros de programação, falhas de hardware) ou por meio de intrusões [Verissimo et al. 2003] (ataques de segurança bem sucedidos, que conduzem o sistema ao comportamento de falha).

Na computação em nuvem grande parte do armazenamento de dados é feito com replicação no intuito de manter a disponibilidade dos dados e melhorar o desempenho da recuperação de informações. O sistema conta com nós primários que recebem as solicitações e coordenam quais nós secundários irão replicar os dados. Esta escolha é feita, normalmente, por ordem sequencial ou aleatória, sem avaliar a condição atual do nó [Greve 2005]. Portanto, é proposto que a partir da chegada de uma requisição de armazenamento, o nó primário efetue uma consulta ao detector de falhas local que irá determinar os nós falhos que não devem ser integrados ao processamento da replicação. O FD é um componente que está presente em todos os nós do sistema, fazendo com que cada processo tenha uma visão adequada dos demais. A Figura 1, abaixo, demonstra a aplicação do FD na nuvem e a visão que cada nó tem do sistema.

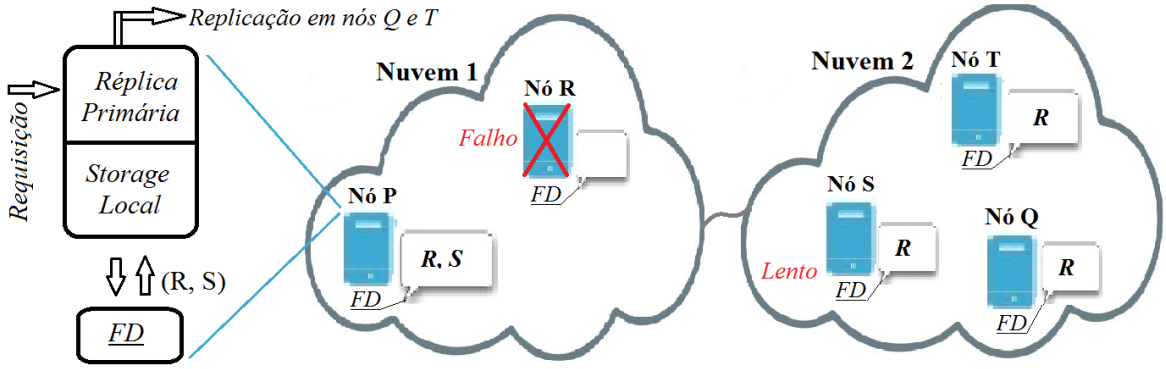

Figura 1. Aplicação do FD no componente de *storage* da nuvem

Modelo do sistema: A rede é constituída de um sistema dinâmico composto por um conjunto finito de processos $\Pi = \{p1, \ldots, pn\}$, com $|\Pi| = n$, e $n > 3$, em cada execução. Este modelo expressa adequadamente as redes dinâmicas onde os nós (máquinas virtuais) entram e deixam a rede livremente [Fan et. al 2012]. Não existe um relógio global conhecido para todos os processos, mas para simplificar a apresentação, assume-se o intervalo T de c*lock* como um conjunto dos números naturais.

Grupo de difusão é conhecido: O protocolo de *broadcast* garante que a difusão ocorre para todos os nós do sistema, conjunto Π, ainda que este valor possa variar entre cada execução. Mesmo os nós recém chegados devem receber a mensagens enviadas por *broadcast*. Portanto, a primitiva *broadcast(m)* invocada pelo processo p, correto, envia uma cópia da mensagem m através do *link* de p para q, para cada $q \in \Pi$ [Greve 2005].

Comunicação *fair-lossy*: Uma mensagem *m* enviada por um processo correto *p* um número infinito de vezes, é recebido pelo processo *q* um número infinito de vezes, ou *q* é bizantino. Como garantia da chegada de mensagens são empregados mecanismos de reconhecimento e retransmissão. Além disso, processos bizantinos não podem interferir na transmissão de mensagens de processos corretos, sendo vetado a duplicação e modificação de mensagens. Processos maliciosos podem tentar criar mensagens falsas, identificando-as como de outro processo. Este problema será assegurado pela autenticação das mensagens. A suposição de canal *fair-lossy* é altamente indicado para a comunicação na internet ou nuvem [Greve 2005].

3.2. Modelo de Falhas dos Processos

Todos os processos estão sujeitos a falhas bizantinas [Lamport et al. 1982], isto é, podem apresentar comportamento arbitrário desviando da execução do algoritmo especificado, podendo parar, omitir o envio e recepção de mensagens, enviar mensagens espúrias, além de trabalhar em conluio com outros processos maliciosos visando corromper o comportamento do sistema. Um processo *p* que não segue a especificação do algoritmo é dito bizantino, caso contrário, ele está correto, neste caso, a função *correct(p)* é verdadeira. Note-se que os conjuntos de processos corretos e falhos formam uma partição.

Falhas bizantinas são divididas em duas classes distintas: quando o comportamento externo de um processo fornece evidências da falha, é chamada de *detectável*, caso contrário, *não detectável* [Kihlstrom et. al. 2003]. Este trabalho lida com falhas detectáveis. Elas são classificadas em falhas de omissão (*progress*), que dificultam o término do cálculo, uma vez que um processo defeituoso não envia as mensagens exigidas pela especificação; e, falhas de segurança (*security*), que violam as propriedades invariantes que os processos devem obedecer de acordo com o algoritmo em execução.

Em sistemas que admitem falhas bizantinas, certos requisitos devem ser atendidos [Kihlstrom et. al. 2003]: *i*) processos corretos devem identificar de forma consistente as mensagens enviadas por cada processo; *ii*) processos corretos devem ser capazes de verificar a validade da mensagem de acordo com os requisitos do algoritmo especificado. Neste trabalho, o primeiro requisito foi satisfeito com a implementação de assinaturas digitais, e o segundo através da utilização de criptografia, como será detalhado a seguir.

Canais autenticados: Assume-se o modelo de criptografia de chave pública e privada, em que cada processo *p* tem um chave privada *K,* com a qual assina suas mensagens, (a exemplo de RSA [Rivest et al. 1978]) e cada outro processo *q* no sistema pode obter a chave pública de *p*, a fim de autenticar o remetente de uma mensagem assinada. Processos bizantinos não podem subverter as primitivas criptográficas.

Validação de mensagens: O algoritmo efetua a validação do conteúdo das mensagens através do modelo de criptografia *hash* (como exemplo, MD5 [Rivest 1992]), evitando falhas arbitrárias. É computacionalmente simples obter o valor *hash* para qualquer mensagem de entrada, além de ser impossível encontrar duas mensagens distintas com o mesmo valor *hash*. Aqui, utiliza-se uma criptografia *hash*, chamado doravante de **H**, com as seguintes propriedades de segurança:

- **H** aceita um dado de tamanho arbitrário como entrada;

- **Ħ** produz uma saída de tamanho fixo, independentemente do tamanho da entrada;
- Dado um valor *hash h*, é computacionalmente difícil de encontrar uma mensagem *M* tal que $h = M$.
- É computacionalmente impossível encontrar qualquer par de mensagens distintas (*M1, M2*) de tal modo que **Ħ**(*M1*) = **Ħ**(*M2*).

3.3. Hipótese de Estabilidade

No desenvolvimento de FDs para redes dinâmicas, um aspecto importante é o período de tempo e as condições em que os processos estão ligados ao sistema. Durante os períodos instáveis, algumas situações, como por exemplo, conexões por períodos muito curtos ou várias ações de entrada e saída de processos ao longo da execução, poderiam bloquear a computação útil da aplicação. Assim, para implementar uma computação global, o sistema deverá apresentar algumas condições de estabilidade: *i*) um processo ao entrar na rede deve interagir com os outros processos para se tornar conhecido; *ii*) um processo deve permanecer na rede o tempo mínimo suficiente para comunicação inicial com os demais processos; *iii*) deve existir pelo menos um processo correto que nunca sai da rede, para que possa ser garantida a propriedade de abrangência do detector: suspeitar de processos falhos permanentemente [Chandra and Toueg 1996].

Assim, considera-se que um processo *p* junta-se à rede em algum momento $t \in T$. Em seguida, *p* deve se comunicar com os outros processos, a fim de ser conhecido (isso pode ocorrer através de uma comunicação *broadcast*). Com a conclusão desta comunicação inicial, todo processo é capaz de se comunicar com *p.* Quando *p* deixa a rede no momento $t' > t$, ele poderá retornar ao sistema no tempo $t'' > t'$ com uma nova identidade, assim, será considerado como um novo processo. Vale ressaltar que ataques sobre a identidade dos processos são validados pelas assinaturas digitais de mensagens.

Definição 1 *(Estados dos processos):*Um processo pode assumir os seguintes estados dentro do sistema:

KNOW: conjunto de processos conhecidos no sistema no tempo *t,* após todos os processos terem entrado no sistema e realizado uma comunicação.

$$KNOW(t) = \forall p, p \in \Pi(t).$$

STABLE: conjunto de processos corretos no sistema no tempo *t*.

$$STABLE(t) = \forall p, p \in \Pi(t) \wedge correct(p).$$

FAULTY: conjunto de processos falhos no sistema no tempo *t*. Esta falha pode ocorrer por parada do processo ou por comportamento bizantino, desviando do algoritmo.

$$FAULTY(t) = \forall p, p \in \Pi(t), \wedge not\ correct(p).$$

3.4. Hipótese de Conectividade

A conectividade do modelo garante as condições mínimas de comunicação entre processos no sistema.

Hipótese 1: no ambiente proposto de computação em nuvem, é garantido que cada processo correto $p \in \Pi$, é capaz de comunicar com todos os outros processos

corretos do conjunto do sistema Π, através dos equipamentos de comunicação da nuvem, independente da presença de processos falhos.

***Hipótese 2**:* determina o limite da tolerância de faltas bizantinas nos processos. Garante que as informações de um processo p vão ser recebidas/envidas para um mínimo de processos corretos na rede. Este limite é de, pelo menos, $f + 1$ processos corretos que podem se comunicar com p. Caso o processo p esteja falho, depois de um período, pelo menos $f + 1$ processos corretos vão suspeitar de p e podem espalhar a suspeita para o restante do sistema, de modo que a propriedade de *abrangência* do FD seja satisfeita (detalhada na próxima Seção). O conjunto de processos do sistema contendo $n > 2f$ processos é suficiente e necessário para a execução correta do protocolo de detecção de falhas bizantinas, como será comprovado no capítulo 5.

3.5. Especificação do Detector de Falhas Bizantino

Baseado na definição de detectores de falhas não confiáveis proposta inicialmente por Chandra e Toueg (1996), Kihlstrom et al. (2003) define as classes de detectores para ambientes com falhas bizantinas, que é o foco deste trabalho. Seja, A um algoritmo que utiliza o detector de falha de um módulo interno. A classe $\Diamond$S (Byz, A) é uma ampliação da classe $\Diamond$S para falhas bizantinas. As propriedades desta classe de detectores são:

Abrangência bizantina forte (A): eventualmente, todos os processos corretos suspeitarão permanentemente de todo o processo que desvia de A;

Exatidão fraca após um tempo*:* eventualmente, um processo correto nunca será suspeito por qualquer outro processo correto.

Definição 2 *($\Diamond S$ (Byz, A) FD não confiável bizantino)*: Seja $t \in T$ e p, j são processos do sistema. Seja $susp_j$ a lista de processos que o processo j suspeita de ter falhado. Assim, os detectores de falhas pertencentes a classe $\Diamond S$ *(Byz, A),* devem manter as seguintes propriedades:

Abrangência bizantina forte (A) $= \{\exists t \in T, \forall t' \geq t, \forall p \in FAULTY \Rightarrow p \in susp_j, \forall j \in STABLE\}$

Exatidão fraca após um tempo $= \{\exists t \in T, \forall t' \geq t, \exists p \in STABLE \Rightarrow p \notin susp_j, \forall j \in STABLE\}$

3.6. Estratégia de Detecção de Falhas

A maioria dos protocolos de detecção de falhas por *crash* é baseada na troca de mensagens do tipo *heartbeats*. Entretanto, em ambientes com falhas bizantinas este mecanismo não é suficiente, pois um processo bizantino pode responder corretamente às mensagens do FD, mas não garantir o progresso e a segurança do algoritmo.

Portanto, utiliza-se aqui uma abordagem livre de tempo para gerar as suspeitas de falhas, apoiando-se num padrão de mensagem requerida pelo algoritmo que não usa qualquer mecanismo de tempo limite de espera de comunicação, chamado de *Query-Response* [Greve 2012]. Assim, um processo p transmite uma mensagem *Query* para os n processos da rede, e aguarda até a recepção de *Response* de pelo menos α processos distintos (α corresponde a quantidade mínima de processos corretos, $\alpha = n - f$, $\alpha \geq f + 1$), e levanta uma suspeita de falha por omissão para os demais processos conhecidos.

Neste ponto, *p* irá enviar uma mensagem de atualização de estado a todos os processos, emitindo seu ponto de vista local sobre as suspeitas.

Esta abordagem considera uma ordem relativa para a recepção de mensagens em que, após um tempo, alguns nós se comunicam mais rapidamente que outros. Deste modo, para garantir a propriedade de *exatidão fraca após um tempo* da classe ◊*S (Byz, A),* deve existir um processo correto *p* cujas mensagens estão sempre entre as primeiras mensagens recebidas em cada grupo de *Response* do algoritmo. Com isso, eventualmente, *p* deixará de ser suspeito por processos corretos [Kihlstrom et. al. 2003].

O módulo de *storage* da computação em nuvem é um sistema distribuído que satisfaz o modelo proposto, pois contém nós primários de alta disponibilidade que recebem as solicitações de armazenamento e replicam em nós secundários [Greve 2005]. Portanto, o nó primário pode representar um componente estável na rede, além disso, alguns nós podem ser considerados mais rápidos, de modo que sempre irão responder mensagens mais rapidamente do que outros nós.

4. Detector de Falhas Bizantinas

Esta Seção descreve o algoritmo (*A*) proposto para detecção de falhas bizantinas em ambiente de nuvem que satisfaz as propriedades do modelo descrito na Seção anterior.

O algoritmo é executado constantemente, enviando por difusão uma mensagem de solicitação, Q*uery*. O intervalo de tempo entre envios consecutivos é finito e arbitrário. Quando o processo *j* recebe uma mensagem Q*uery* de um processo *p*, *j* lhe confirma a recepção com uma mensagem *Response*. O algoritmo é composto por três comportamentos principais, são eles:

Geração de suspeita de falha: as suspeitas geradas por falhas de omissão estão ligadas a mensagens requeridas pelo algoritmo *A*. Assim, o processo *j* é suspeito de falhar pelo processos *p*, se *j* não enviar as mensagens que deveria para *p* de acordo com *A*. Para tanto, cada mensagem deve ter um identificador distinto. Além das suspeitas locais, um processos também adota suspeitas se receber mensagens de *Suspicion* devidamente validadas de, pelo menos, $f + 1$ remetentes diferentes. Esta exigência não permite que processos bizantinos criem suspeitas sobre processos corretos. As mensagens *Suspicion* são responsáveis por espalhar na rede o resultado das suspeitas internas de cada processo. De acordo com as hipóteses de conectividade, todos os processos corretos recebem as mensagens, ou seja, pelo menos $f + 1$ processos, assim as suspeitas poderão ser adotadas, garantindo a propriedade *abrangência bizantina forte* da classe do FD.

Geração de erro de suspeita: Seja *j* um processo suspeito de não enviar uma mensagem *m* requerida por *A*. Se, após um tempo, um processo *p* recebe *m* de *j* devidamente assinada e validada, *p* irá declarar o erro de suspeita e espalhar uma mensagem *Suspicion* para que os demais processos possam fazer o mesmo. Este comportamento assegura a propriedade de *exatidão fraca após um tempo* da classe do FD. Entretanto, também dá a possibilidade a um processo malicioso criar uma suspeita e revogá-la continuamente, degradando o desempenho do detector de falha. No entanto, não há como distinguir esta situação de um processo lento ou de instabilidade no canal.

Identificação de falha bizantina: Para assegurar a correção da detecção de falhas, deve-se estabelecer um padrão para as mensagens de *A*. Estas podem ser do tipo *Response* (reposta a uma mensagem inicial *Query*) ou *Suspicion* (mensagem de

atualização de suspeitas internas). Como todas as mensagens do algoritmos são assinadas através de criptografia de chave pública e seu contudo é validado através de criptografia *hash*, é perfeitamente possível garantir que uma mensagem *m* esteja formatada corretamente, e que seu remente é autentico. Dessa maneira, *m* é válida se: *i) m* possui conteúdo correto, ou seja, computado o seu valor *hash* é igual ao valor *hash* recebido; *ii) m* segue adequadamente um padrão especificado pelo algoritmo, isto é, está sintaticamente correta com os tipos *Response* ou *Suspicion*. Seguindo estes princípios, se um processo correto detecta a invalidade de uma mensagem recebida, ele irá suspeitar permanentemente do remetente, como processo bizantino, e irá espalhar uma mensagem de *Suspicion* para os processos restantes, para que a suspeita seja propagada.

4.1. Descrição do Algoritmo de Detecção de Falhas Bizantinas

A seguinte notação é utilizada na implementação do algoritmo:

- *output*: guarda a saída do FD, o conjunto de processos identificados como suspeitos de terem falhado;
- *know*: conjunto de processos conhecidos pelo FD. É atualizado a cada mensagem correta recebida dos tipos *Response* ou *Suspicion*;
- *mistake*: array que guarda os processos com erro de suspeita. O array é indexado pelo identificador do processo *p* e armazena a mensagem correta que confirma o erro na suspeita de *p*;
- *inter_susp:* conjunto de processos suspeitos internamente. Uma suspeita interna é gerada por não receber uma mensagem *Response* requerida, ou pela presença de pelo menos $f + 1$ suspeitas externas de um processo;
- *exter_susp*: array que armazena as suspeitas externas (geradas pelos outros nós). O array é indexada pelo identificador de cada processo *p*, e armazena o conjunto de processos dos quais *p* suspeita;
- *byzantine:* array que guarda os processos considerados bizantinos. O array é indexado pelo identificador do processo *p* e armazena a mensagem incorreta que confirma *p* como bizantino;
- *received:* conjunto de processos dos quais recebeu mensagens *Response*;
- *Braodcast(m):* transmite uma mensagem *m* para todos os processos da rede*;*
- *Send(m, j):* transmite a mensagem *m* para o processo *j*;
- $H(m)$*:* função de criptografia *Hash* que dá como saída o valor *hash* da mensagem *m*;
- $K_{Priv(p)}$*:* função de criptografia de chave privada do processo *p;*
- $K_{Pub(p)}$*:* função de criptografia de chave pública do processo *p;*

O algoritmo ainda utiliza os seguinte procedimentos auxiliares:

- *InternalSusp(j):* adiciona as suspeitas por omissão geradas pelo FD, assim o processo *j* é adicionado em *inter_susp* e *output* (linhas 34-36);
- *Mistake(j, m):* insere erros de suspeitas de omissão anteriores, desta forma, o processo *j* é adicionado no grupo de *Mistake*, juntamente com a mensagem que confirma o erro, removendo-o dos grupos de suspeita interna e externa. Caso este processo não foi anteriormente identificado como bizantino, também é retirado do *output* (linhas 38-44);
- *Byzantine(j, m):* adiciona as suspeitas obtidas através da validação da mensagem, o processo *j* é adicionado permanentemente em *byzantine* e *output,* juntamente com a mensagem *m* que confirma tal ação (linhas 45-47);

- *ValidateReceived(j, m):* decodifica e valida a mensagem *m* recebida. Inicialmente é validada a segurança (*ValidateHash*) da mensagem, identificando processos bizantinos. Caso a mensagem *m* esteja correta, de acordo com seu tipo, *Response* ou *Suspicion,* é inserido o erro de suspeita do processos ou é efetuada a atualização do estado do FD, respectivamente (linhas 49-59);
- *UpdateState(j, m):* atualiza o estado do FD local com os dados recebidos. Processos contidos em *byzantine* e *Mistake* de *j* tem a mensagem que gerou tal conclusão reavaliada, processos contidos em *inter_susp* de *j* são inseridos como suspeita externa. Neste momento é avaliado se existe pelo menos *f + 1* suspeitas externas de um processo (*exter_susp*), e então, assume-se este como um suspeito interno (linhas 61-73);
- *ValidateHash (j, m):* valida a segurança da mensagem *m* recebida. Decodifica *m* com a chave pública de *j* ($K_{Pub(\ j\ )}$) efetuando a verificação do remetente e valida do conteúdo e formato de *m*, podendo avaliar *j* como bizantino (linhas 75-90).

A implementação completa do algoritmo *A* está descrita abaixo. Cada processo *p* executa paralelamente as quatro tarefas que compõe o algoritmo:

Tarefa T1 (Envio de solicitação inicial) (linhas 5-6). O FD envia constantemente uma mensagem do tipo *Query* em *broadcast*, e aguarda o envio da mensagem *Response* pelos demais processos. Esta mensagem não precisa ser criptografada ou assinada.

Tarefa T2 (Envio de resposta) (linhas 8-10). Ao receber uma mensagem do tipo *Query*, o processo retorna uma mensagem *Response* ao remetente. É calculado o valor *Hash* da mensagem e criptografado com sua chave privada (*KPriv*) para garantir a autenticidade da mensagem, resultando na mensagem *CR* (linha 9).

Tarefa T3 (Gerando suspeitas de falhas) (linhas 12-24). Após o envio uma *Query* (tarefa T1), o algoritmo espera pela recepção de α mensagens *Response* de remetentes distintos, que são armazenados em *received* (linha 15). Todos os outros processos conhecidos são adicionados no conjunto de falhas de omissão (linhas 16-17). Em seguida, cada mensagem recebida é verificada pela função *ValidateReceived()*(linhas 19-20). Mensagens não validadas geram entrada em processo falhos (bizantinos), atualizando a saída do detector, *output* (linha 47). Mensagens corretas, geram erros nas suspeitas de falha anteriores (linha 54). Após todo processo é enviado via *broadcast* a mensagem *Suspicion* de atualização de estado para os outros processos, composta por (*byzantine, mistake, inter_susp*), de forma segura, calculando o valor Hash da mensagem, e criptografando com sua chave privada (*KPriv*) (linhas 22-24).

Tarefa T4 (Recebimento de mensagens *Response* atrasadas e mensagens de atualização de estado *Suspicion*) (linhas 26-31). Quando uma mensagem *m* é recebida de um processo *j*, a mensagem é validada através da função *ValidateReceived()* (linha 28). Assim, há duas possibilidades: *i*) *m* é uma mensagem *Response* que foi recebida provavelmente após a suspeita do processo *j* gerada pela tarefa T3. Neste caso, *m* é tratada de forma similar à tarefa *T3*; *ii*) *m* é uma mensagem de atualização de estado *Suspicion*. Neste caso, a mensagem contém as opiniões de *j* sobre o estado da rede (*byzantine, mistake, inter_susp*) para a atualização do estado do FD local (linha 56). Finalmente, é enviado via *broadcast* a mensagem *Suspicion*, devidamente criptografada, para propagar as novas informações do detector (linha 29-31).

Algoritmo *A* – Detector de Falhas Bizantinas Assíncrono

```
1: Init:
2: output ← known ← inter_susp ← ∅
3: exter_susp ← mistake ← byzantine ← [ ]
4:
5: Task T1:  /* Solicitação inicial - Query */
6:  Broadcast(Q)
7:
8: Task T2:  /* Envio de Response para j */
9:  CR = K_Priv(p) {H(Response) + Response}
10:  Send(CR, j)
11:
12: Task T3:  /* Gera novas suspeitas */
13:  when (requires messages Response) do
14:  wait until receive m from α distinct
     processes j
15:    received ← j
16:  for all p ∈ ( known \ received ) do
17:     InternalSusp(j)
18:  end for
19:  for all m received from j do
20:     ValidateReceived(j, m)
21:  end for
22:  S ← SUSPICION(byzantine, mistake,
     inter_susp)
23:  CS ← K_Priv(p){ H(S) + S }
24:  Broadcast(CS)
25:
26: Task T4:  /* Recebe mensagens Suspicion
    ou Response de processos lentos*/
27:  upon receipt of m from j do
28:     ValidateReceived(j, m)
29:  S ← SUSPICION(byzantine, mistake,
    inter_susp)
30:  CS ← K_Priv(p){ H(S) + S }
31:  Broadcast(CS)
32:
33: /* PROCEDIMENTO AUXILIARES */
34: procedure InternalSusp( j ):
35:  inter_susp ← inter_susp ∪  { j }
36:  output ← output ∪ { j }
37:
38: procedure Mistake( j, m ):
39:  mistake[ j ]  ← mistake[ j ] ∪ { m }
40:  inter_susp ← inter_susp \ { j }
41:  exter_susp ← exter_susp[ ] \  { j }
42:  if  byzantine[ j ]  = ∅ then
43:     output ← output  \ { j }
44:  end if
45: procedure Byzantine( j, m ):
46:  byzantine[ j ] ← byzantine[ j ] ∪ { m }
47:  output ← output ∪ { j }
48:
49: procedure ValidateReceived( j, m ):
50:  mr ← ValidateHash( j, m )
51:  if mr != ∅ then
52:     known ← known ∪ { j }
53:     if j ∈ inter_susp then
54:        Mistake( j, m )
55:     end if
56:     if mr is SUSPICION then
57:        UpdateState( j, mr)
58:     end if
59:  end if
60:
61: procedure UpdateState( j, m ):
62:  for all ( j_x, m_x ) ∈ m.byzantine do
63:     ValidateReceived( j_x, m_x )
64:  end for
65:  for all ( j_x, m_x ) ∈ m.mistake do
66:     ValidateReceived( j_x, m_x )
67:  end for
68:  for all ( j_x ) ∈ m.inter_susp do
69:     exter_susp[ j ] ← exter_susp[ j ] ∪ { j_x }
70:     if | ( j_x ) ∈ exter_susp[ ] | > f then
71:        InternalSusp(j_x)
72:   end if
73:  end for
74:
75: function ValidateHash( j, m):
76:  md ← K_Pub(j) { m }
77:  if md is [ Hmsg, msg ] then  /* remetente */
78:     Hmc ← H( md.msg )
79:     if Hmc != md.Hmsg  then  /* conteúdo */
80:        Byzantine( j, m )
81:        return ∅
82:     else
83:        if (md.msg = Response) or  /* padrão */
             (md.msg = Suspicion) then
84:           return md.msg
85:        else
86:           Byzantine( j, m)
87:           return ∅
88:        end if
89:     end if
90:  end if
```

5. Provas de Correção do Algoritmo

Para que o detector de falhas apresentado na Seção 4 pertença a classe $\Diamond S$ *(Byz, A),* deve satisfazer as propriedades de *abrangência bizantina forte* e *exatidão fraca após um tempo*. A seguir, é descrito um esboço das provas do algoritmo.

5.1. Abrangência Bizantina Forte

Lema 1: Se um processo *p* nunca enviar a mensagem *m*, então, um processo $j \in STABLE$ nunca irá executar *Mistake*(*p*, *m*), para retirar *p* de *output*.

Prova: Suponha, por contradição, que um processo correto *j* executa *Mistake*(*p*, *m*). Assim, o procedimento *ValidateReceived()* foi executado (linha 54). Este procedimento é invocado em dois casos: *1)* na recepção de mensagens *Response* e *Suspicion*, tarefa T3 (linha 20) e T4 (linha 28); *2)* na atualização de estado interno com informações externas (*UpdateState()*, linhas 63 e 66). Em todos os casos, a autenticidade e validade da mensagem *m* são verificadas (*ValidateHash()* linhas 75-90). Portanto, um processo defeituoso não pode enviar *m* no lugar do *p*. Já no caso *2)*, não é possível gerar uma chamada de *Mistake()* se, na mensagens *Suspicion,* não existir uma mensagem de *p* a ser validada por *ValidateReceived()*. Por fim, conclui-se numa contradição em todos os casos, uma vez que para *m* ser recebido, *p* deve envia-la.

Lema 2: Seja *p* um processo que falha por omissão, $p \in FAULTY$. Então, eventualmente, todos os $j \in STABLE$ irão incluir permanentemente *p* em *output*.

Prova: Seja *t* o tempo em que *p* falha por omissão, $correct^{t}(p)$ é falsa. Seja $t' < t$ o tempo que *p* entra na rede $known^{t'}(p)$ é verdadeira. Seja, $t' \leq s \leq t$ é o momento em que processo *p* não envia a mensagem *m* do tipo *Response*. Sempre que o algoritmo *A* requer um *m*, *j* irá aguardar recepção de *m* por α processos distintos (linhas 13 e 14). Este predicado é satisfeito pela *Hipótese 2* ($|\Pi| > 2f$). Se *p* não enviar *m*, não será adicionado em *received* (linha 15). Assim, com execução das linhas 16-17 e 35-36, *p* será incluído em *inter_susp* e em *output*. Se *p* falha por omissão, nunca enviará posteriormente a mensagem *m*. Assim, de acordo com *Lema 1, p* não será removido de *inter_susp* e *output* (linhas 38-44).

Lema 3: Seja *p* um processo que falha por segurança, $p \in FAULTY$. Então, eventualmente, todos os $j \in STABLE$ irão incluir permanentemente *p* em *output.*

Prova: Seja *t* o tempo em que *p* falha por segurança, $correct^{t}(p)$ é falsa, pelo envio de uma mensagem *m* não validada por *A*. Sendo a comunicação autenticada e a mutação de mensagens excluída do modelo, *m* foi recebida em algum momento nas linhas 14 de T3 ou 27 de T4. Em ambos os casos será invocado *ValidateReceived()* (linhas 20 e 28) que por sua vez faz a chamada de *ValidateHash()* (linhas 50*)*. Esta função irá atestar a invalidade de *m* pela avaliação do conteúdo e formato (linha 79 e 83). Assim, *Byzantine()* (linhas 80 e 86) adiciona *p* em *byzantine* e *output*. A partir destas condições, e, sendo que, *p* só é removido de *output* se não houver nenhum entrada de *p* em *byzantine* (linhas 42-43), *p* é definitivamente adicionado a *output*.

5.2. Exatidão Fraca Após Um Tempo

Lema 4: Se $p, j \in STABLE$, então *j* nunca chama *Byzantine*(*p*, -).

Prova: O procedimento *Byzantine()* é invocado apenas nas linhas 80 e 86 em *ValidateHash()*. Sabendo-se que *j* é correto, esta chamada ocorria apenas se *p* enviasse uma mensagem não validada, sendo isso impossível, uma vez que *p* também é correto, assim, suas mensagens serão avaliadas positivamente (linhas 79 e 83). Ainda que um processo malicioso enviasse uma mensagem como *p*, o processo *j* a descartaria, pois todas as mensagens são devidamente autenticadas (linha 77).

Lema 5: Seja $p \in STABLE$. Após um tempo nenhum processo $j \in STABLE$ irá chamar *InternalSusp(p)*.

Prova: O procedimento *InternalSusp()* é chamado em dois casos: *1)* após a recepção de mensagens *m* do tipo *Response* em T3 (linha 17) e *2)* no procedimento *UpdateState()* (linha 71), quando o processo *j* recebe as suspeitas externas de *p*. No primeiro caso, sendo *t* o tempo que $correct^{t}(p)$ é verdadeira, *j* receberá *m* de *p*, $\forall t' \geq t$, se $j \in STABLE$. Em seguida, adiciona *p* em *received* (linha 15) e assim, *j* não invoca *InternalSusp(p)* na linha 17. O segundo caso só é alcançado por $j \in STABLE$ na atualização de suspeitas externas (linhas 68-73). Isto significa que, cada suspeita externa de um processo correto foi primeiramente gerada como uma suspeita interna (linhas 70 e 71). A partir do mesmo argumento do caso *1)*, um processo correto *j* nunca adiciona *p* a *inter_susp em* $t' \geq t$ na execução de T3 (linha 17). Entretanto, um processo bizantino $b \in FAULTY$ pode incluir *p* em seu *inter_susp*. Todavia, a quantidade *f* de processos falhos no sistema não é capaz de validar o predicado da linha 70. Assim, o processo correto *j* não invocará *InternalSusp(p)* na linha 71.

Lema 6: Seja *p*, $j \in$ STABLE e *m* uma mensagem *Response* de *p*. Se $p \in$ *inter_susp* de *j*, então, após um tempo, *j* invocará *Mistake(p, m)*.

Prova: Sendo *p* um processo correto que utiliza canais *fair-lossy* autenticados, eventualmente *j* receberá *m* de *p* (devidamente assinado e validado) (linha 27). De acordo com a hipótese do lema, $p \in$ *inter_susp* de *j,* assim, com a execução das linhas 20, 50 e 53, sendo *p* correto, *j* irá chamar Mistake(*p, m*) (linha 54).

Lema 7: Seja $p \in$ STABLE. Após um tempo, $\forall j \in$ STABLE tal que $p \notin$ *output* de *j*.

Prova: Pelo *Lema 4*, em algum $t' \geq t$, $p \notin$ *output* de *j*. Pelo *Lema 5, j* não adiciona *p* em *output* por uma chamada de *InternalSusp(p)*. É possível que em $t'' < t$, mensagens *Response* requerias por *j* tenham inserido *p* em *inter_susp* de *j*. No entanto, pelo *Lema 6*, em algum momento futuro, *j* realizará a chamada de *Mistake(p, m)*, retirando assim, *p* em *inter_susp* de *j* (linha 40). Em conseguinte, como não existirá valor em *byzantine[p]*, *Lema 4, p* será removido do *output* de *j* (linha 43).

6. Conclusões e Trabalhos Futuros

Neste artigo é apresentado um detector de falha bizantina da classe $\Diamond S$ *(Byz, A),* que traz as seguintes características inovadoras para a aplicação nos componentes de *storage* da computação em nuvem: *i)* é adequado a redes dinâmicas, de forma a promover a escalabilidade e capacidade de adaptação inerentes ao sistema em nuvem; *ii)* trata o sistema com assíncrono, retirando as primitivas temporais na detecção de falhas, e modela a realidade de nuvens federadas e *Mashups*. Como trabalho futuro, pretende-se ampliar o protocolo para realidade de outros módulos da computação e nuvem, bem

como implementar a avaliação de desempenho frente a outros modelos detectores de falhas.

Referências

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

Fan G., Yu H., Chen L. and Liu D. (2012) "Model Based Byzantine Fault Detection Technique for Cloud Computing" In *IEEE Services Computing Conference (APSCC)*, Guilin, Asia-Pacific, pp. 249-256.

Ganesh, A., Sandhya, M. and Shankar, S. (2014) "A study on fault tolerance methods in Cloud Computing", In *Advance Computing Conference (IACC), IEEE International,* pp. 844 - 849

Garraghan P., Tounend P. and Jie X. (2011) "Byzantine fault-tolerance in federated cloud computing". In *Proccessing of 6th International Symposium on Service Oriented System Engineering (SOSE 2011). IEEE Computer Society*, pp. 280-285.

Greve, F. (2005) "Protocolos Fundamentais para o desenvolvimento de Aplicações Robustas" SBC-SBRC, Brasil, pp. 330-398.

Greve, F., Lima M., Arantes L. and Sens P. (2012) "A Time-Free Byzantine Failure Detector for Dynamic Networks" In *IEEE Dependable Computing Conference (EDCC),* Sibiu, Ninth European. pp.191 – 202.

Kihlstrom K. P., Moser L. E. and Melliar-Smith P. M. (2003) "Byzantine Fault Detectors for Solving Consensus," The Computer Journal, vol. 46, no. 1, pp. 16–35

Lamport L., Shostak R. and Pease M., (1982) "The Byzantine generals problem" ACM Transactions on Programming Languages and Systems, vol. 4, pp. 382–401.

Poledna S. (1996) "Fault Tolerant Real-Time Systems: The problem of replica Determinism*".* The Springer International Series in Engineering and Computer Science.

Ramasamy H. and Cachin. C. (2005) "Parsimonious Asynchronous Byzantine-Fault-Tolerant Atomic Broadcast" In *Proc. 9th International Conference on Principles of Distributed Systems*, Berlin. Germany

Rivest R., Shamir A. and Adleman L, (1978) "A method for obtaining digital signatures and public-key cryptosystems," Commun. ACM, vol. 21, pp. 120–126.

Rivest R. (1992), "The MD5 Message-DigestAlgorithm". MIT Laboratory for Computer Science and RSA DataSecurity.

Sivakami, R. and Nawaz G. (2011) "Reliable communication for MANETS in military through identification and removal of byzantine faults" In *IEEE 3rd International Conference on Electronics Computer Technology (ICECT).* Kanyakumari, Indian, vol. 5, pp. 377-381.

Verissimo, P., Neves, N. F., et. al. (2003) "Intrusion-Tolerant Architectures: Concepts and Design", Lemos, R., Gacek, C., Romanovsky, A. (eds), Architecting Dependable Systems, v. 2677, LNCS, Springer-Verlag.