Armazenamento Seguro em Nuvem com Detecção de Falhas Bizantinas

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. *Failure detectors are essential services for designing fault tolerant applications as they provide information about system process failures. In modern distributed systems that have inherently dynamic behavior, such as cloud computing, the design of these services is a challenge, especially in the treatment of byzantine failures. Thus, this paper aims at proposing a secure 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 failure detection. Simulation experiments have validated the presented approach.*

Resumo. *Detectores de falhas são serviços essenciais para o projeto de aplicações tolerantes a falhas, pois provêm informações sobre falhas de processos do sistema. Nos sistemas distribuídos modernos que possuem comportamento inerentemente dinâmico, dentre eles a computação em nuvem, a concepção destes serviços é um desafio, principalmente no tratamento de falhas bizantinas. Por tanto, 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. A solução foi implementada e os resultados de simulação são apresentados como forma de validar a abordagem apresentada.*

1. Introdução

A computação em nuvem é um dos grandes cenários da computação distribuída moderna que evolui para integração de sistemas dinâmicos e auto-organizáveis. Ela propõe um novo paradigma para a gestão de infraestrutura, oferecendo possibilidades para implantar diversos recursos em ambientes distribuídos através de um modelo de *utility computing*. Este gerenciamento de nuvem permite o provisionamento de recursos computacionais sob demanda, operado através de máquinas virtuais em *data centers* de provedores distribuídos [Ganesh et al. 2014].

Neste contexto, novos recursos computacionais são criados em tempo de execução com base em solicitações existentes, derivando 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]. Assim, os protocolos distribuídos clássicos tornam-se 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 [Garraghan et al. 2011].

Uma forma de manter a confiabilidade desses sistemas distribuídos é dispor de de mecanismos que os tornem tolerantes a falhas, com recursos que identifiquem e implementem formas de contorno. Na resolução destes problemas a principal dificuldade encontrada é a complexidade de se detectar as falhas existentes no sistema. Assim, o detector de falhas (*FD*) é um serviço fundamental capaz de ajudar no desenvolvimento de sistemas distribuídos tolerantes a falhas, trabalhando com um oráculo distribuído que 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 implementa 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].

O serviço de armazenamento (*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, tais como a sincronia de visões e o acordo [Greve 2005]. Assim, identificar os processos falhos e evitar sua utilização é essencial para garantir o desempenho do sistema de nuvem.

A replicação de dados na nuvem pode ser feita de forma estática, com um grupo de processos corretos predeterminados na rede, ou dinâmica, onde o grupo de processos é determinado a partir da solicitação e com base em critérios adotados [Greve 2005]. Estes dois modelos utilizam tipicamente um FD para controlar os nós disponíveis para replicação. Neste contexto, encontra-se sistemas de *storage* distribuídos largamente utilizados como o Dynamo da empresa Amazon [DeCandia et al. 2007] e o Cassandra da empresa Facebook [Lakshman and Malik 2009]. Este último, emprega um sistema chamado Zookeeper [Hunt et al. 2010] para controlar a vivacidade dos nós na distribuição do armazenamento.

Grande parte das implementações de detectores de falhas são baseadas numa comunicação entre todos os nós, em que é monitorada a atividade dos nós através da troca periódica de mensagens *heartbeats* e do *timeout* da comunicação. [Hunt et al. 2010]. Porém, avançando neste contexto, alguns trabalhos foram propostos para lidar com dinamismo da rede adotando uma abordagem livre de tempo (não utiliza *timeout*) para efetuar a detecção de falhas em sistemas dinâmicos, assumindo um ambiente assíncrono. Estes artigos se destacam ainda por trabalharem com detecção de falhas malignas (bizantinas) no sistema [Greve et al. 2012, Matos and Greve 2015].

Neste contexto, este trabalho tem o intuito de contribuir com o estudo sobre a detecção de falhas bizantinas para a computação em nuvem com a aplicação, implementação e validação de um algoritmo livre de tempo como detector de falhas no

serviço de *storage* na nuvem. Ele é uma continuidade da pesquisa desenvolvida em [Matos and Greve 2015] e visa apresentar uma simulação da solução proposta em ambiente de nuvem. O FD é utilizado como um oráculo que informa um grupo de processos falhos na rede, incapazes de atender as requisições de armazenamento e replicação com segurança. Para efeito de avaliação, utilizamos o serviço Zookeeper [Hunt et al. 2010] como padrão de comparação para o algoritmo proposto. A comparação tem o duplo propósito de comparar o desempenho das abordagens, além de confrontar duas categorias de detectores: o FD livre de tempo (aqui proposto) e o FD baseado em timeouts (integrado ao Zookeeper).

As principais contribuições deste trabalho são a implementação, simulação e validação de um novo algoritmo de detecção de falhas proposto em [Matos and Greve 2015], com sua subsequente aplicação para o armazenamento confiável da computação em nuvem.

O restante deste artigo está organizado da seguinte forma: A Seção 2 discute os principais trabalhos relacionados. A Seção 3 apresenta o modelo de ambiente do detector de falhas. Na Seção 4 é apresentado o algoritmo. A Seção 5 apresenta a avaliação da proposta. Por fim, as conclusões e trabalhos futuros na Seção 6.

2. Trabalhos Relacionados

Na literatura, encontra-se trabalhos relacionados a detectores de falhas aplicados a varias áreas da computação, na sua maioria, implementações ligadas à falhas benignas e ambiente síncronos. Outro seguimento de trabalhos é focado nas falhas malignas, ou bizantinas, que propõe soluções baseadas no envio de *heartbeats*, através controle do *timeout* da comunicação [Ramasamy and Cachin 2005]. Neste sentido, tem-se a proposta de Sivakami and Nawaz (2011) que apresenta um protocolo de roteamento tolerante a falhas bizantinas. Ela implementa uma busca de melhor caminho através da utilização de um detector de falhas malignas, operando em cada nó do sistema.

Ainda no contexto de falhas bizantinas, encontra-se um novo grupo de trabalhos que aplica 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. Avançando neste contexto, tem-se um grupo de trabalhos que propõe a adoção de uma abordagem livre de tempo para efetuar a detecção de falhas bizantinas em sistemas dinâmicos aplicada em redes móveis [Greve et al. 2012].

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.

Em [Matos and Greve 2015], apresentamos um novo algoritmo de detecção de falhas para o armazenamento da computação em nuvem, que tem como características inovadoras: (i) a adoção de uma estratégia de detecção de falhas sem a utilização de mecanismos de tempo (*time free*), baseado em ambiente assíncrono, modelando assim, a

dinamicidade dos sistemas em nuvem; *(ii)* detecção de falhas na presença de agentes maliciosos, o que constitui um desafio em sistemas distribuídos.

O presente trabalho dá continuidade à proposta trazida por [Matos and Greve 2015] e apresenta os resultados da aplicação e simulação da detecção de falhas bizantinas na computação em nuvem, assumindo as fracas prerrogativas temporais de um sistema assíncrono.

3. Modelo para Detector de Falhas Bizantinas no Armazenamento em nuvem

Nesta seção é discorrido sobre como detector de falhas proposto está inserido no módulo de armazenamento da nuvem, bem como detalhes do ambiente e de seu comportamento.

3.1. Modelo de Nuvem Assíncrono

A computação em nuvem é composta de *data centers* dispersos geograficamente que efetuam a computação e o armazenamento dos dados, uma realidade ainda mais evidenciada assumindo a utilização de nuvens federadas ou nuvens *Mashups* [Garraghan 2011]. A 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 respostas 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, *iii*) não existem suposições sobre a velocidade relativa dos processos [Fan et. al 2012].

Na computação em nuvem a comunicação entre todos os módulos heterogêneos que compõem o sistema é dada pela troca de pacotes através da internet, estando sujeita a atrasos arbitrários e perdas [Ganesh et al. 2014]. Isto provê mais duas propriedades ao modelo: *iv*) não existem limites temporais para a transferência da mensagem[1], *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].

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 *data centers* 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).

[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.

3.2. Arquitetura do Sistema

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 escolhem, de forma sequencial ou aleatória, quais nós secundários irão replicar os dados [Greve 2005]. Essa característica pode ser observada na maioria dos sistemas de armazenamento utilizados em larga escala. Como exemplo tem-se o banco de dados distribuído Cassandra [Lakshman and Malik 2009], que propaga o dado armazenado em um nó, em N réplicas no sistema. As réplicas podem ser escolhidas de forma sequencial dentro do grupo de nós, ou serem designadas por um componente externo chamado Zookeeper [Hunt et al. 2010], que para detecção de falhas efetua uma comunicação par a par com todos os nós do sistema, e assim define a escolha. Ainda neste contexto tem-se o sistema de armazenamento Dynamo [DeCandia et al. 2007] que trabalha de forma semelhante ao sistema citado anteriormente, mas para controle do grupo de nós do sistema implementa um detector de falhas baseado em *heartbeats* através de um protocolo *gossip*, obtendo informações sobre os nós ativos no sistema. Outro exemplo é o banco de dados distribuído Bigtable [Chang et al. 2006], criado pela empresa Google para armazenar distribuidamente uma grande quantidade de dados. Internamente o sistema utiliza o protocolo Paxos para coordenar integridade das informações na presença de falhas de processo. Este protocolo tem como base um detector de falhas não confiável para identificar a situação de cada nó da rede.

Partindo deste contexto, é proposto que o detector de falhas implementado neste trabalho seja utilizado como um componente interno do sistema armazenamento que aponta diretrizes de escolha dos nós a serem utilizados na replicação. Isto se dá da seguinte maneira: a partir da chegada de uma requisição de armazenamento no nó primário, será efetuado uma consulta ao detector de falhas local que irá determinar os nós falhos que não devem ser integrados no 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 proposto na nuvem e a visão que cada nó tem do sistema.

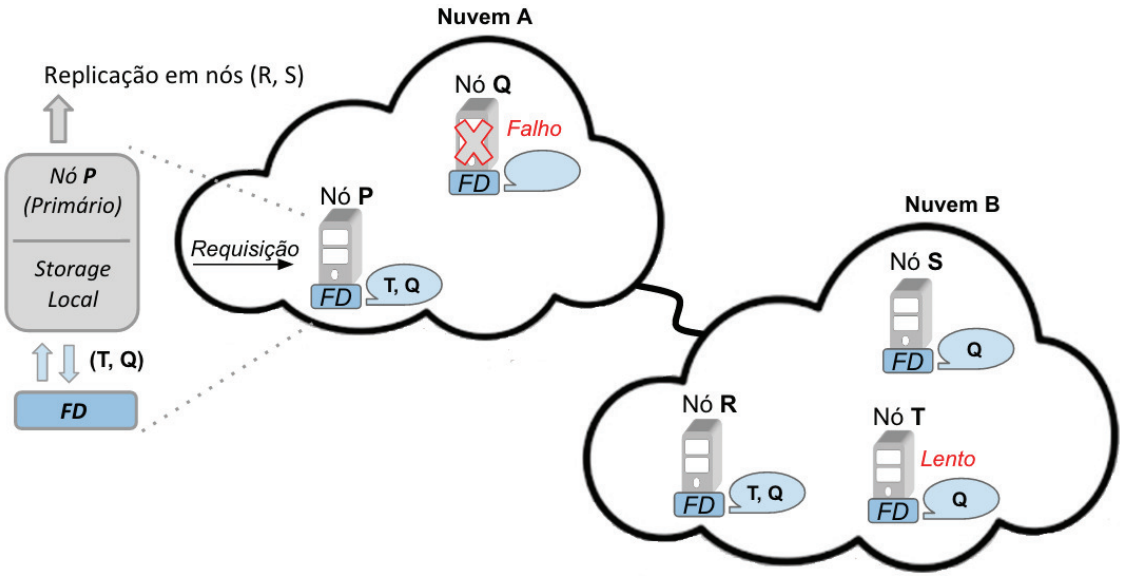


Figura 1. Aplicação do FD no componente de *storage* da nuvem

3.3. Modelo do Sistema

Considera-se um sistema distribuído dinâmico constituído 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.

Na transmissão de mensagens entre processos, 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].

Para o ambiente de computação em nuvem proposto, adota-se a comunicação do tipo *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 confirmação 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 [Greve 2005].

3.4. Modelo de Falhas dos Processos

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. Os conjuntos de processos corretos e falhos formam uma partição.

Na literatura, as 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*), onde o 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.

Alguns requisitos devem ser atendidos em sistemas que admitem falhas bizantinas [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 utilização de canais autenticados, e o segundo através da criptografia, como 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;
- **H** 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 **H**(*M1*) = **H**(*M2*).

Limite de falhas de processos: Para garantir conectividades dos processos, deve-se delimitar um limite de tolerância de falhas bizantinas nos processos. Isto garante que as informações de um processo *p* vão ser recebidas/enviadas 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. 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.

4. Detector de Falhas Bizantinas

Esta Seção descreve o algoritmo (*A*), proposto em [Matos and Greve 2015], que utiliza o detector de falhas como módulo interno para efetuar a identificação de falhas bizantinas, satisfazendo o 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*. As mensagens *Query* e *Response* determinam aqui o padrão de mensagens trocadas pela aplicação de armazenamento. 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 processo *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 processo 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.

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.

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* (resposta 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 conteúdo é validado através de criptografia *hash*, é perfeitamente possível garantir que uma mensagem *m* esteja formatada corretamente, e que seu remente é autêntico. 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. 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 para os processos restantes.

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;
- *know:* conjunto de processos conhecidos pelo FD;
- *mistake:* array que guarda os processos com erro de suspeita;
- *inter_susp:* conjunto de processos suspeitos internamente;
- *exter_susp*: array que armazena as suspeitas externas (geradas pelos outros nós);
- *byzantine:* array que guarda os processos considerados bizantinos;
- *received:* conjunto de processos dos quais recebeu mensagens *Response*;
- *Broadcast(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 (linhas 34-36);
- *Mistake(j, m):* insere erros de suspeitas de omissão anteriores (linhas 38-44);
- *Byzantine(j, m):* adiciona as suspeitas de processos bizantinos (linhas 45-47);
- *ValidateReceived(j, m):* decodifica e valida a mensagem *m* recebida (linhas 49-59);
- *UpdateState(j, m):* atualiza o estado do FD local (linhas 61-73);
- *ValidateHash (j, m):* valida a segurança da mensagem *m* recebida (linhas 75-90).

A implementação do algoritmo *A* é composta por quatro tarefas executadas paralelamente por cada processo: a tarefa T1, realiza o envio de solicitações iniciais (linhas 5-6); a tarefa T2, realiza o envio de respostas (linhas 8-10); a tarefa T3 é responsável pela geração inicial de suspeitas de falhas (linhas 12-24); e por fim a tarefa T4 efetua o recebimento de mensagens para a atualização do estado do detector (linhas 26-31).

As provas de correção deste algoritmo, bem como suas especificações de acordo

com a classificação de detectores de falhas não confiáveis proposta Chandra e Toueg (1996), Kihlstrom et al. (2003) podem ser obtidas em Matos and Greve (2015).

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 j ∈ ( 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 l ( j_x ) ∈ exter_susp[ ] l > 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. Implementação e Avaliação de Desempenho do Detector de Falhas

Nessa Seção é apresentado um estudo de avaliação de desempenho da implementação do detector de falhas assíncrono proposto. Os resultados obtidos são comparados com os apresentados pelo detector existente no Zookeeper [Hunt et al. 2010]. O intuito dessa análise comparativa é fazer um contraponto entre o detector de falhas assíncrono criado e um detector baseado em temporização largamente utilizado em aplicações reais existentes. Em seguida é avaliado o detector de falhas na presença de nós bizantinos.

5.1. Modelo de Simulação

Esta simulação busca avaliar a qualidade de serviço dos detectores de falhas através de algumas métricas baseadas em tempo sugeridas por Chen and Toueg (1996). No ambiente de *storage* em nuvem proposto, o detector de falhas deve estar pronto a responder às solicitações das réplicas primárias e informar os nós falhos da rede não indicados para replicação. Assim, quanto mais rápido e correto forem identificados os nós falhos, melhores serão as respostas emitidas pelo detector, consequentemente ocasionando um serviço de replicação mais seguro.

Vale salientar que o ambiente simulado foi escolhido para eliminar da avaliação as possíveis interferências dos ambientes reais, com o proposito de obter resultados mais precisos sobre o modelo avaliado. Todos os experimentos foram realizados em um computador Intel Core i5 3.2GHz com 8GB de RAM e sistema operacional Ubuntu 14.04 64bits. Os detectores de falhas foram implementados na linguagem Java utilizando o pacote de *sockets* para a troca de mensagens no algoritmo assíncrono proposto, e o pacote Apache Zookeeper[2] versão 3.4.9 na implementação do Zookeeper. Cada cenário do teste de desempenho foi repetido por 10 vezes obtendo uma média dos dados das execuções como resultado final. Neste contexto, cada processo equivale a um nó da rede.

Detector de Falhas Zookeeper: Este detector é baseado numa conexão cliente servidor, onde é armazenada em cada servidor uma variável de conexão de cada nó cliente chamada de zNode. Assim, cada servidor periodicamente envia uma mensagem de *heartbeat* a seus clientes verificando a atividade destes nós. Todos os nós implementam o comportamento de servidor para obter informações sobre os nós da rede, e também de cliente para responder a solicitações dos outros nós. Este detector trabalha unicamente com falhas benignas. Para questões de implementação de detecção de falhas foi utilizado modo *standalone* com zNodes tipo *ephemeral*.

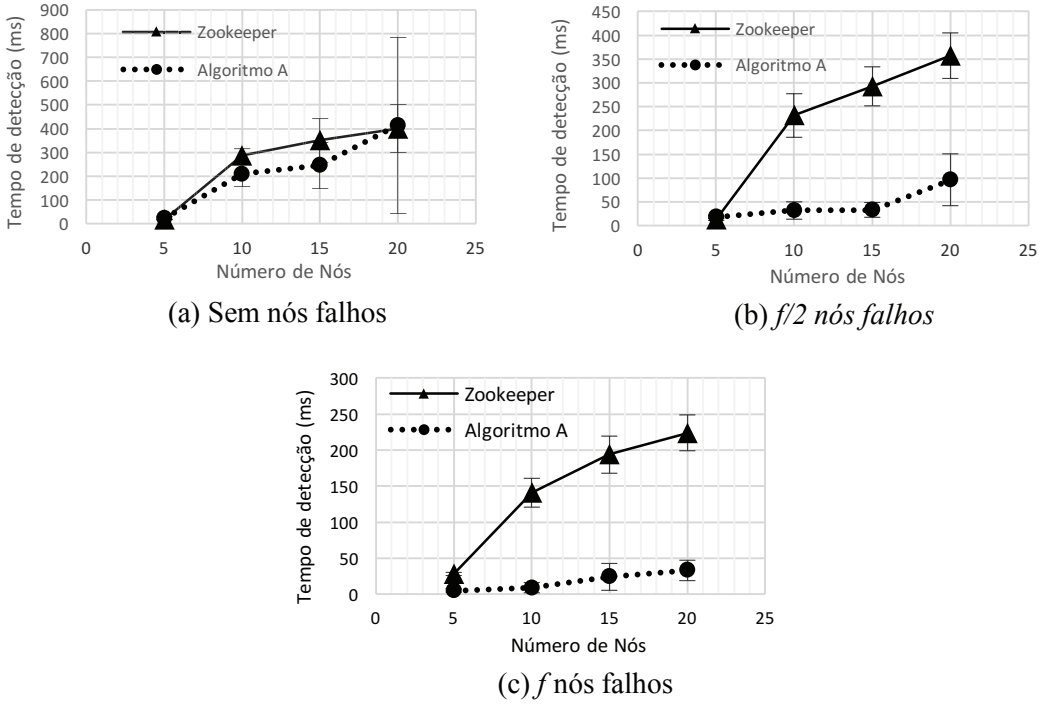
Detector de Falhas Assíncrono: Na implementação deste detector foi utilizado como tempo de reenvio de mensagens tipo *Query*, executado pela tarefa **T1**, o tempo de sessão padrão do Zookeeper, *TickTime* com 3 segundos, permitindo que os dois detectores trabalhem com os mesmos parâmetros. Com o intuito de evitar sobrecarga na rede, ao receber uma mensagem tipo *Suspicion,* a mensagem a ser repassada em *broadcast* não é reenviada ao remetente. Também é implementado número de sequência de mensagens para validar a sua atualidade. Vale ressaltar que essas otimizações não interferem na correção do algoritmo.

[2] Disponível em http://zookeeper.apache.org

5.2. Avaliação da Detecção de Falhas Benignas

Para avaliar a qualidade de serviço de ambos os detectores, foi mensurado o impacto da variação da densidade de nós na rede e os respectivos tempos de detecção de falha dos dois detectores (Figura 2). O tempo de detecção de falha consiste no intervalo entre o instante *t* em que a falha aconteceu, e o instante t' > t em que ela é permanentemente detectada por todos os processos corretos. Para a avaliação foram criados três cenários com diferentes números de nós falhos: *i*) rede sem nós falhos, *ii)* rede com *f*/2 nós falhos, e *iii)* rede com *f* nós falhos, sendo *f* o limite de tolerância de falhas de acordo com a quantidade de nós da rede N, $f = (N/2) - 1$. As falhas são inseridas em nós aleatórios, sendo que em cada execução existe um numero fixo de nós falhos.

No cenário com apenas nós corretos, o tempo de detecção foi considerado como o momento em que todos os nós reconhecem a não existência de falhas na rede.



(a) Sem nós falhos

(b) *f/2 nós falhos*

(c) *f* nós falhos

Figura 2. Tempo de detecção de falhas benignas

De acordo com a Figura 2 é possível perceber que quando não existe falha na rede, os dois detectores se comportam de maneira semelhante, com o crescimento do tempo de detecção associado à densidade de nós da rede. Já no momento que são inseridos nós falhos, (b) e (c), o algoritmo proposto consegue imprimir uma redução considerável no tempo de detecção, diferenciando-se do detector Zookeeper. Isto se justifica porque o detector de falhas assíncrono utiliza o princípio de suspeitar de todos os processos que não estiverem entre as primeiras α respostas recebidas (α corresponde a quantidade mínima de processos corretos). Assim independente de um tempo de espera de respostas, estes processos já são tidos como falhos para o detector. Erros de suspeitas para processos com respostas atrasadas são tratados posteriormente com a chegada de cada mensagem. Outro fator que influi para o baixo tempo de detecção são as mensagens do tipo *Suspicion,* que espalham na rede as informações sobre os

processos falhos descoberto por cada nó. Estas mensagens proporcionam, além de uma melhoria no tempo detecção, uma garantia para que toda a rede suspeite dos nós falhos.

5.3. Avaliação da Recuperação de Falsas Suspeitas

Como o detector proposto permite gerar falsas suspeitas sobre nós da rede e efetua correções posteriores, faz-se necessário validar o impacto deste mecanismo na detecção e falhas. Nesta avaliação foi verificado o tempo de recuperação de falsas suspeitas, sendo ele o período entre o tempo *t,* onde o detector suspeita erroneamente de um processo, e o tempo t' > t, onde esta suspeita é eliminada. Utilizou-se os mesmos três cenários de falhas da avaliação anterior.

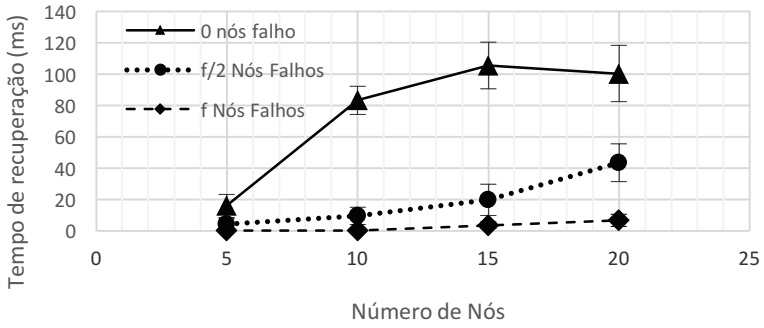


Figura 3. Tempo de recuperação a falsas suspeitas do detector proposto

A Figura 3 mostra que na rede sem nós falhos o tempo de recuperação de falsas suspeitas é consideravelmente maior, pois são geradas várias suspeitas de nós corretos mais lentos, que são corrigidas posteriormente. Entretanto nos cenários com nós falhos, a recuperação é efetuada de forma rápida, seja pela chegada das respostas dos nós suspeitos, reavaliando o processo como ativo, ou pela chegada de mensagens do tipo *Suspicion,* que confirmam a situação de falhas nos processos da rede. Enfim, nos três casos avaliados, o tempo de recuperação de falhas não é fator limitante para a qualidade da detecção de falhas, pois equivale apenas cerca de 30% do tempo total de detecção, apresentado na Figura 2.

Destaca-se ainda que mesmo não trabalhando com *timeouts* de conexão, o detector proposto nunca irá suspeitar de um grupo de processos com comunicação mais rápida na rede, sempre que suas mensagens estiverem dentro das primeiras α respostas recebidas.

5.4. Avaliação da Detecção de Falhas Bizantinas

Nesta avaliação os processos bizantinos agem de forma randômica entre três comportamentos: como um processo correto, emitindo mensagens corretas; como um processo falho, com omissão de mensagens; ou de forma arbitrária, se desviando do padrão de mensagens exigida pelo algoritmo. O detector de falhas proposto é capaz de identificar os três comportamentos do processo, entretanto é garantido que um processo uma vez identificado como bizantino, sempre seguirá com esta classificação, mesmo que este volte a se comportar como um processo correto.

Para esta avaliação do detector, foi mensurado o impacto no tempo de detecção dos nós bizantinos com a variação da densidade da rede (Figura 4). O tempo de

detecção bizantina consiste no intervalo entre o instante *t* em que um nó bizantino emitiu sua primeira mensagem bizantina, e o instante t' > t em que este nó é permanentemente detectado por todos os processos corretos. Na avalição foi utilizado apenas os dois cenários com falhas, pois a rede com ausência de falhas já foi avaliada anteriormente.

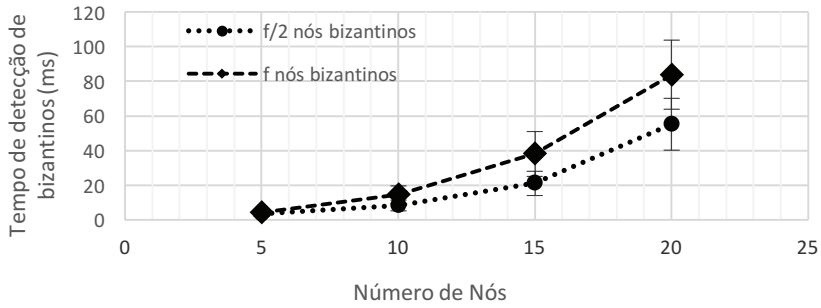


Figura 4. Tempo de detecção de falhas bizantinas

A partir da Figura 4 é possível perceber que após emissão de uma mensagem bizantina, a identificação e propagação desta informação pela rede é feita de forma rápida pelas mensagens tipo *Suspicion.* Salienta-se que para uma informação ser tida como verdadeira, deve ser recebida de pelo menos $f+1$ processos, para que a união de nós bizantinos não possa resultar numa subversão da rede.

Fazendo contraponto entre as Figuras 2 e 4, avalia-se que a detecção de falhas bizantinas acontece, em alguns casos, em menor tempo do que detecção de falhas benignas. Isto se justifica porque a suspeita de falhas bizantinas pode ocorrer na primeira mensagem recebida, desde que essa seja emitida por um nó bizantino, já as suspeitas de falhas benignas seguem o padrão do algoritmo de surgirem após α respostas recebidas.

6. Conclusões e Trabalhos Futuros

Neste artigo é apresentado uma proposta de armazenamento seguro na nuvem utilizando como coordenador de replicação um detector de falhas bizantinas assíncrono. O detector traz as seguintes características inovadoras aplicadas à computação em nuvem: *i)* adequado a redes dinâmicas, de forma a promover a escalabilidade e capacidade de adaptação; *ii)* trata o sistema com assíncrono, retirando as primitivas temporais na detecção de falhas, modelando a realidade de nuvens federadas e *Mashups*. Os resultados da avalição de desempenho puderam validar a eficácia do detector proposto. Por fim, como continuação deste trabalho pretende-se ampliar o protocolo para realidade de outros módulos da computação em nuvem.

Referências

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

Chang, F., Dean, J., Ghemawat, S. et. Al. (2006) "Bigtable: a distribuited storage system for structured data". In Porc of 7h Conf. USENIX. Berkeley, CA.

DeCandia G., Hastorun D., Jampani M., et. Al. (2007) "Dynamo: amazon's highly available key-value store". *In Proccessing of 21th ACM SIGOPS symposium on Operating systems principles*. pp. 205–220..

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,* 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.

Hunt, P., Konar, M., Junqueira, F. and Reed B., (2010) "Zookeeper: Wait-free coordination for internet-scale systems In *Proc. 10th USENIX Annual Technical Conference (USENIXATC)*, Boston, USA.

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

Lakshman A. and Malik P. (2009) "Cassandra – A Decentralized Structured Storage System"

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

Matos C. and Greve F., (2015) "Um Detector de Falhas Bizantinas Assíncrono Aplicada à Computação em Nuvem" XVI WTF-SBRC, Vitória, ES, Brasil. SBC.

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 Inter Conference on Electronics Computer Technology (ICECT)*. Kanyakumari, Indian, 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.