SMIT: Uma Arquitetura Tolerante a Intrusões Baseada em Virtualização

Valdir Stumm Júnior[1], Lau Cheuk Lung[1], Miguel Correia[2], Joni da Silva Fraga[3], Jim Lau[3]

[1]Departamento de Informática e Estatística
Universidade Federal de Santa Catarina (UFSC) – Florianópolis – Brasil

[2]Faculdade de Ciências
Universidade de Lisboa – Lisboa – Portugal

[3]Departamento de Automação e Sistemas
Universidade Federal de Santa Catarina (UFSC) – Florianópolis – Brasil

{stummjr,lau.lung}@inf.ufsc.br, mpc@di.fc.ul.pt,

{fraga,jim}@das.ufsc.br

Resumo. *Soluções práticas para o desenvolvimento de aplicações distribuídas tolerantes a faltas bizantinas tem sido alvo de pesquisas nos últimos anos. Tais soluções visam oferecer resistência aos sistemas contra ataques de todos os tipos, inclusive maliciosos, tornando-os assim tolerantes a intrusões. Recentemente, o uso de virtualização para construção de um ambiente confiável tem sido considerado por alguns desses trabalhos. Este artigo apresenta SMIT, uma arquitetura tolerante a intrusões que toma proveito de uma área de memória compartilhada entre máquinas virtuais residentes em uma máquina singular para simplificar o protocolo de consenso. O artigo também apresenta uma abordagem distribuída para SMIT, que utiliza um modelo híbrido de falhas.*

1. Introdução

A confiabilidade de sistemas computacionais se tornou um tema bastante discutido nas últimas décadas. Isso se deve, principalmente, à crescente participação desses sistemas em atividades de suporte básico ao cotidiano de nossa sociedade. Sistemas financeiros, de controle de geração e distribuição de energia, de controle de dispositivos hospitalares, são alguns exemplos de sistemas computacionais que as organizações vêm utilizando. Para que a sociedade possa depositar sua confiança sobre tais sistemas, é necessário que estes sejam resistentes, de modo a preservar seu funcionamento correto, mesmo sob a presença de faltas.

Dada a importância de tais sistemas para o correto funcionamento de serviços básicos da sociedade, várias abordagens foram propostas. Dentre tais abordagens, a Replicação Máquina de Estados (RME) Tolerante a Faltas Bizantinas (BFT) tem sido alvo de diversas pesquisas há mais de vinte e cinco anos [Lamport et al. 1982, Schneider 1990]. No entanto, somente na última década é que surgiram propostas que apresentam viabilidade prática para tal abordagem [Castro and Liskov 2002, Yin et al. 2003, Correia et al. 2004, Kotla et al. 2007, Luiz et al. 2008].

Protocolos BFT, em geral, são compostos pela replicação de um serviço através de um grupo de máquinas que se comunicam de modo a oferecer um serviço replicado que atenda aos requisitos de confiabilidade, integridade e disponibilidade, que são fundamentais para que se obtenha Confiança no Funcionamento (*Dependability* [Avizienis and Kelly 1984]). Em geral, os estudos recentes com propostas para BFT visam diminuir o número de restrições impostas por tais protocolos. A definição de arquiteturas híbridas com uso de componentes seguros (os *wormholes* [Correia et al. 2007]) ou premissas de sincronia mais práticas e realistas são alguns dos métodos utilizados. Redução do número de passos de comunicação para alcançar o acordo [Kotla et al. 2007] e otimização do uso de recursos computacionais disponíveis [Yin et al. 2003, Correia et al. 2004, Luiz et al. 2008] são preocupações constantes nos trabalhos mais recentes. Além disso, as soluções também procuram circunscrever algumas impossibilidades teóricas (p. ex. condição FLP [Fischer et al. 1985] e a necessidade de, no mínimo, $3f + 1$ réplicas para tolerar faltas bizantinas [Lamport et al. 1982]).

Recentemente, alguns trabalhos propuseram o uso de tecnologias de virtualização para implementar sistemas tolerantes a intrusões utilizando uma única máquina física [Reiser and Kapitza 2007, Stumm Júnior et al. 2009]. Uma vantagem aparente dessa abordagem é a diminuição do custo de replicação através da utilização de somente uma máquina hospedando várias máquinas virtuais. A maior desvantagem é que a máquina única constitui um ponto singular de falhas.

O presente artigo apresenta SMIT (*Shared Memory based Intrusion Tolerance*), uma arquitetura tolerante a intrusões, bem como um protocolo de consenso para a execução de serviços sobre a arquitetura replicada. Nesse trabalho mostramos que é possível simplificar o protocolo de consenso BFT através do uso de uma área de memória compartilhada para comunicação entre máquinas virtuais, evitando a passagem de mensagens pela rede para comunicação entre estas. A simplicidade do protocolo de consenso foi um dos principais objetivos perseguidos por este trabalho, visto que protocolos complexos levam a implementações complexas, mais suscetíveis a faltas de projeto e desenvolvimento [Nagappan et al. 2006].

Este trabalho dá continuidade ao estudo apresentado anteriormente [Stumm Júnior et al. 2009], trazendo consigo as seguintes contribuições: suporte à presença de clientes maliciosos, suporte a canal de comunicação não-confiável entre clientes e réplicas, um novo protocolo para consenso entre as réplicas de serviço baseado em uma idéia de líder **oportunista**, proposição de uma abordagem distribuída, a qual reverte a incapacidade do trabalho anterior tolerar faltas de *crash* no sistema anfitrião.

2. Modelo de Sistema

Nosso modelo é composto por três tipos de sistemas: os **clientes**, que são entidades externas ao ambiente virtualizado, o sistema **anfitrião**, que pode ser representado por um Monitor de Máquinas Virtuais (VMM - *Virtual Machine Monitor*) ou por um sistema operacional sobre o qual um VMM está sendo executado, e os sistemas **convidados**, que são máquinas virtuais executando as réplicas de serviço sobre o VMM. O sistema anfitrião fornece uma abstração de memória compartilhada para os sistemas convidados, a qual é utilizada pelas réplicas de serviço para a troca de mensagens visando implementar um protocolo de consenso tolerante a faltas bizantinas. Essa abstração de memória compar-

tilhada será descrita em mais detalhes na Subseção 2.2, dada sua relevância para o nosso trabalho.

Assumimos um sistema assíncrono, composto por clientes e servidores. Clientes e servidores são conectados através de uma rede, a qual pode perder, atrasar, reordenar e corromper os pacotes. O conjunto de servidores, $S = \{S_0, S_1, ..., S_{n-1}\}$, possui no mínimo $2f + 1$ sistemas virtuais executando sobre uma única máquina física, ou seja, $|S| \geq 2f + 1$. No máximo f dentre os membros de S podem apresentar comportamento bizantino. Os clientes, aqui representados pelo conjunto $C = \{C_0, C_1, ...\}$, são sistemas que enviam requisições para os servidores através da rede. Não há limites no número de clientes bizantinos em nosso modelo, uma vez que um cliente bizantino não compromete a corretude do sistema. O comportamento bizantino significa que o processo pode se comportar de forma diferente da especificação do protocolo, ou seja, pode parar sua execução, se omitir e produzir mensagens inconsistentes.

Técnicas de diversidade de projeto [Avizienis and Kelly 1984] são aplicadas no desenvolvimento das réplicas de serviço. A idéia principal por trás disso é que as réplicas devem falhar independentemente umas das outras, ou seja, a falha de uma réplica não implica diretamente na falha de outra. Tal diversidade pode ser aplicada em nível de sistema operacional e de aplicação.

Assumimos que o sistema anfitrião pode apresentar vulnerabilidades, porém estas não podem ser exploradas. Possíveis atacantes seriam máquinas virtuais comprometidas, ou mesmo entidades externas acessando diretamente o anfitrião. Para evitar o primeiro tipo de atacante, confiamos que o VMM forneça isolamento adequado entre as VMs e o anfitrião. O segundo tipo de atacante pode ser evitado bloqueando acesso direto pela rede ao sistema anfitrião, o que pode ser obtido utilizando técnicas de *firewall* ou desabilitando/removendo as interfaces de rede do anfitrião.

2.1. Segurança

Em nossa proposta, são necessárias algumas preocupações com a segurança da informação do sistema. A primeira questão relevante com relação a segurança é a *Integridade* das informações trocadas entre os membros do sistema. Para que o sistema mantenha as propriedades que garantem sua correção, é necessário que uma mensagem enviada por um membro c seja entregue a outro membro s com conteúdo idêntico ao que foi remetido por c, ou seja, a mensagem não pode ter seu conteúdo alterado indevidamente. Outra preocupação importante com relação a nossa proposta diz respeito à *Autenticidade*, ou seja, se o remetente de uma mensagem é realmente quem alega ser. Assim, é importante um mecanismo que possibilite que um membro c, ao receber uma mensagem alegadamente oriunda de s, verifique se esta mensagem realmente foi enviada por s. Além da integridade e autenticidade, é importante que exista um mecanismo de *Controle de Acesso*, para garantir que os clientes somente executem nas réplicas as operações para as quais estão autorizados, evitando também que réplicas maliciosas obtenham êxito propondo alterações em dados relativos ao serviço dos clientes. A questão da *Confidencialidade* da informação não é tratada neste trabalho.

Para garantir as propriedades acima em todas as operações realizadas no sistema, aplicamos mecanismos de criptografia assimétrica [Rivest et al. 1978] ao nosso modelo. Para isso, todas as mensagens trocadas por clientes e réplicas são assinadas utilizando

técnicas de criptografia assimétrica. Assim, para cada par comunicante $\{m_0, m_1\}$, onde m_0 e m_1 são membros do conjunto de clientes e do conjunto de réplicas, respectivamente, m_0 deve conhecer a chave pública de m_1 e vice-versa. Caso contrário, ambos não serão capazes de checar a integridade e autenticidade das mensagens trocadas entre eles. O mecanismo de controle de acesso deve ser implementado de acordo com as políticas da aplicação a ser executada sobre a arquitetura, podendo utilizar para isso a assinatura da mensagem para realizar a autenticação.

Nosso modelo assume que um atacante não possui poder computacional para quebrar as técnicas criptográficas utilizadas.

2.2. *Postbox*

Conforme descrito anteriormente, nossa arquitetura toma proveito de uma abstração de memória compartilhada, a qual será referida como *postbox*. Esse componente é utilizado pelas réplicas de serviço (VMs) para troca de mensagens entre si, de forma que ao escrever um valor nesta, *a réplica está difundindo atomicamente tal valor para todas as réplicas corretas*. Qualquer réplica pode gravar valores na memória compartilhada para serem lidos pelas outras réplicas, porém, o ritmo no qual cada réplica pode escrever valores nesta é limitado, de modo a evitar ataques de negação de serviço. Todas as réplicas corretas no conjunto S lêem todos os valores na mesma ordem em que foram escritos na *postbox*. A *postbox* possui uma interface simples, composta por dois métodos:

- *append(value) : boolean*
- *read() : value*

O método ***append*** grava um valor na *postbox*, juntamente com um identificador da réplica que o gravou (o VMM gerencia isso, então não há falsificação sobre o remetente da mensagem). O valor é armazenado em uma posição imediatamente após o valor gravado na última operação *append* que foi realizada. O valor de retorno é do tipo booleano, indicando se a operação foi realizada com sucesso ou não. O método ***read*** retorna o valor imediatamente posterior ao valor retornado na última operação *read* executada pela réplica. Ou seja, os valores são lidos e escritos em modo FIFO (*First In, First Out*).

Um detalhe crucial para nossa abordagem é que a *postbox* deve operar em modo *append-only*, ou seja, uma vez lá armazenado, um valor não pode ser alterado. Essa restrição evita a situação na qual uma réplica maliciosa escreve um valor, aguarda até que uma ou mais réplicas realizem a leitura, e então altera tal valor, de modo que as réplicas que ainda estão por realizar a leitura correspondente leiam valores diferentes dos lidos pelas primeiras. Tal situação poderia gerar uma inconsistência em todo o serviço replicado.

É importante perceber que a principal vantagem da utilização da *postbox* é que esta é um componente único e centralizado na arquitetura, assim os conjuntos de operações de leitura executados por todas as réplicas possuem a mesma ordem, ou seja, todas as réplicas corretas possuem uma visão homogênea do conteúdo da *postbox*. Então, para qualquer conjunto de operações W_i contendo *k* operações de escrita, que vão da *n-ésima* até a *(n+k)-ésima* posição da *postbox*, todas as réplicas corretas que realizarem um conjunto de *k* operações de leitura a partir da posição *n* possuirão um conjunto de leituras R_i, com conteúdo idêntico a W_i.

A *postbox* deve ser fornecida pelo VMM para os sistemas convidados. O VMM deve fornecer mecanismos básicos de controle de acesso e deve empregar algoritmos de escalonamento no acesso à memória compartilhada que sejam justos o suficiente de forma que todas as máquinas virtuais consigam acessar tal componente, o mais rápido possível, evitando ataques de negação de serviço por parte de VMs corrompidas. A Figura 1 ilustra uma visão geral da arquitetura aqui descrita.

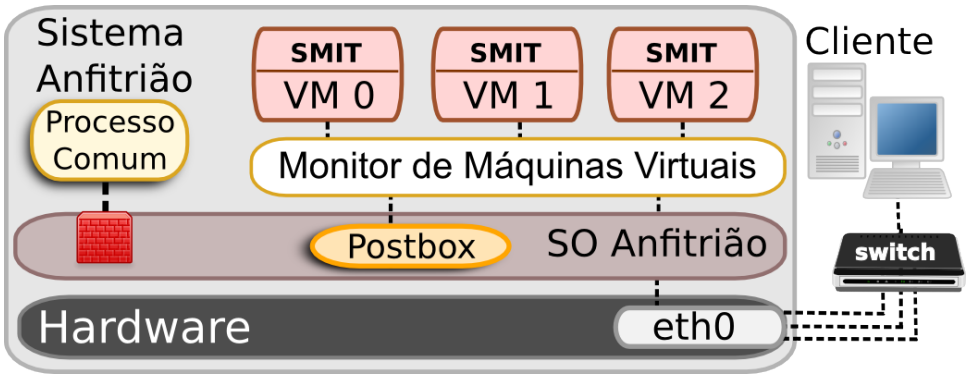

Figura 1. Visão Geral da Arquitetura SMIT

2.3. DSMIT

Sistemas replicados virtualmente em uma única máquina física possuem a limitação de serem vulneráveis a faltas de *crash* no sistema físico. Ao ocorrer um *crash* no anfitrião, todas as réplicas de serviço (executando sobre máquinas virtuais) também sofrem o *crash*. Para contornar tal limitação, apresentamos uma abordagem com replicação tolerante a faltas de *crash* para o sistema anfitrião, chamada de DSMIT (*Distributed* SMIT).

Para tanto, modificamos nosso modelo de sistema, replicando o sistema anfitrião fisicamente e passando a utilizar um modelo de falhas híbrido [Correia et al. 2002]. Nesse modelo, as réplicas de serviço (VMs) são capazes de tolerar faltas bizantinas enquanto que os sistemas físicos (anfitriões) toleram apenas faltas de *crash*. Assim, introduzimos em nosso modelo a replicação física de anfitriões e a noção de grupos de réplicas. Um grupo de réplicas é um conjunto formado por todas as réplicas de serviço residentes em uma mesma máquina física. Para cada sistema anfitrião que executa SMIT (permanecendo com as mesmas premissas feitas anteriormente), existe um grupo de, no mínimo, $2f + 1$ réplicas de serviço executando sobre ele. Cada anfitrião executa uma instância de SMIT e os anfitriões são conectados entre si através de um canal de comunicação confiável privado síncrono, por onde farão a sincronização de seus estados (*postboxes*) locais.

A abordagem DSMIT requer, no mínimo, $f_c + 1$ máquinas físicas para tolerar f_c faltas de *crash* em nível de anfitrião. Cada máquina física deve ter pelo menos $2f_b + 1$ máquinas virtuais executando SMIT juntamente com o serviço. f_b é o número máximo de membros bizantinos tolerados por máquina física. De forma geral, são necessárias $(2f_b + 1) \times (f_c + 1)$ máquinas virtuais no total. A estratégia de replicação, bem como seu funcionamento e detalhes serão discutidos na Seção 4.

3. Protocolo de Consenso

Para executar sobre a arquitetura descrita na seção anterior, propusemos um protocolo de consenso Tolerante a Faltas Bizantinas (BFT) para que seja possível a execução de

serviços Tolerantes a Intrusões sobre esta. A presente seção apresenta as propriedades que o protocolo deve cumprir, bem como os detalhes do protocolo em si.

3.1. Propriedades do Protocolo

O protocolo BFT aqui apresentado deve garantir as seguintes propriedades para prover um serviço correto:

- ***Safety***: todas as réplicas corretas executam as mesmas requisições na mesma seqüência.
- ***Liveness***: requisições submetidas por clientes corretos terão sua execução completada em algum momento, ou seja, não importa o que aconteça, o protocolo sempre faz progresso.

Para garantir a propriedade *safety* na abordagem Replicação Máquina de Estados, as requisições enviadas pelos clientes devem ser todas recebidas (*Acordo*) e executadas na mesma ordem (*Ordem Total*) por todas as réplicas corretas [Schneider 1990]. Além disso, o serviço provido pelas réplicas corretas deve apresentar comportamento determinístico. As provas de correção estão disponíveis na versão estendida do artigo[1].

3.2. Detalhes do Protocolo

Esta seção descreve o conjunto de algoritmos que juntos compõem o protocolo BFT a ser executado na arquitetura detalhada na Seção 2. Sucintamente, nosso protocolo funciona da seguinte maneira:

1. Cliente c envia uma requisição req para as réplicas do conjunto S;
2. As réplicas do grupo S recebem req e alguma réplica r_i é a primeira a propor a execução de req, escrevendo uma proposta de execução para req, a qual inclui a mensagem inteira, na *postbox*;
3. Todas as réplicas corretas de S leem a proposta feita por r_i na *postbox*, executam a mensagem na ordem proposta e enviam a resposta diretamente para c;
4. c aguarda até que receba $f + 1$ respostas de conteúdo idêntico para req.

Como pode-se perceber analisando a seqüência de passos detalhada acima, em nosso protocolo não existe uma figura de líder fixo do protocolo. O líder, membro que propõe a ordem de execução das requisições, pode variar de requisição para requisição. Tradicionalmente, protocolos para BFT utilizam a idéia de um líder fixo, responsável por propor as ordens, o qual perde o cargo apenas quando as demais réplicas desconfiarem de que este se comporta de forma suspeita [Castro and Liskov 2002, Kotla et al. 2007, Stumm Júnior et al. 2009]. Outras propostas apresentaram a idéia de alteração do líder do protocolo a cada conjunto de mensagens para o qual este define uma ordem, ou seja, após definir a ordem para um conjunto de mensagens, é feita a mudança de líder [Veronese et al. 2009]. Em nossa abordagem, não existe a figura de líder fixo e tampouco é aplicada a rotatividade na escolha do líder, pois o líder relativo a uma requisição será a réplica que primeiro propor sua execução através da *postbox*, caracterizando assim o que chamamos de *líder oportunista*. É importante ressaltar que uma réplica maliciosa não tem poder de intereferir na corretude do protocolo. Através das assinaturas, eliminamos a

[1]Disponível em: http://www.inf.ufsc.br/~stummjr/smit-full.pdf

possibilidade de uma réplica se fazer passar por um dos clientes (a não ser que este último tenha divulgado sua chave privada para tal réplica).

O Algoritmo 1 mostra o lado cliente do protocolo. Como mostra a linha 1, a mensagem é enviada para as réplicas pertencentes a S com um certificado contendo uma assinatura, gerada utilizando a chave privada de c, anexada a mensagem. Após enviar a requisição para as réplicas, o cliente passa a aguardar por $f + 1$ requisições assinadas corretamente e com conteúdos (campos $data$ e ts) idênticos. Assim, é garantido que pelo menos uma das $f + 1$ mensagens é oriunda de uma réplica correta, validando todas as $f + 1$ mensagens recebidas.

Algoritmo 1 Envio de requisição do cliente para o grupo de réplicas S

```
1: multicast_send(⟨REQ, data, ts, cert⟩, S)
2: repeat
3:     buffer ← buffer ∪ recv()
4: until f + 1 matching replies ∈ buffer
```

Algoritmo 2 Tarefas de recebimento de requisições e leitura de propostas

```
task receive_requests( ):
1: loop
2:     req ← receive()
3:     if req.t > cache[req.id].t then
4:         if signed(req) then
5:             postbox.append(⟨PROP, req, id⟩)
6:         end if
7:     else if req.t = cache[req.id].t then
8:         reply ← cache[req.id]
9:         send(⟨REPLY, reply, id, req.t, sgn⟩, rep.id)
10:    end if
11: end loop

task read_proposals( ):
12: loop
13:    prop ← postbox.read()
14:    if prop = ⟨PROP, req, id⟩ then
15:        if prop.req.t > cache[prop.req.id].t and signed(prop.req) then
16:            reply ← execute(prop.req.data)
17:            cache[prop.req.id] ← reply
18:            send(⟨REPLY, reply, prop.req.t, id, sgn⟩, prop.req.addr)
19:        end if
20:    end if
21: end loop
```

O lado servidor do protocolo é composto por dois fluxos básicos de execução que são executados concorrentemente. O primeiro, descrito na tarefa

`receive_requests()` (linhas 1-11) do Algoritmo 2 é responsável por receber as requisições do cliente e propor a execução destas na *postbox*. Esse fluxo do servidor permanece aguardando por mensagens dos clientes. Ao receber uma mensagem, este testa se a mensagem ainda não foi processada. Caso já tenha sido executada, ele verifica se a resposta para tal mensagem ainda está em sua *cache* e retorna a resposta ao cliente sem reprocessá-la (linhas 8 e 9), visto que esta requisição pode ter sido gerada devido a uma retransmissão decorrente de atrasos na rede. Caso seja uma mensagem nova, o servidor verifica a assinatura da mensagem e, caso esteja corretamente assinada, escreve na *postbox* uma proposta de execução para esta (linha 5).

O segundo fluxo de execução, descrito na tarefa `read_proposals()` do Algoritmo 2, é responsável por ler propostas da *postbox* e executá-las, caso sejam corretas. Tal fluxo permanece realizando leituras na *postbox* em busca de propostas para execução de novas requisições. Ao ler uma proposta, a réplica verifica se a mensagem correspondente não foi previamente processada (linha 15). Caso já tenha sido executada, o servidor simplesmente ignora a proposta. Caso contrário, verifica a assinatura da mensagem (linha 15). Se estiver corretamente assinada, o servidor executa a mensagem (linha 16), adiciona ela em sua lista de mensagens já processadas (linha 17) e envia a resposta ao cliente (linha 18).

O lado servidor do protocolo necessita que as réplicas mantenham em seu estado interno uma tabela indexada pelo identificador do cliente, representada no algoritmo pela variável *cache*. Tal tabela contém, em cada posição, a resposta para a última mensagem processada cujo remetente possua como identificador o índice de tal posição. Ela é usada para possibilitar a retransmissão de mensagens que o cliente pode não ter recebido por atrasos ou falhas na rede. Um mecanismo de coleta de lixo baseado em temporização deve ser usado sobre as entradas de tal tabela para evitar que o tamanho desta ultrapasse a capacidade de memória da máquina.

4. Tolerando Faltas de *Crash* no Anfitrião

Conforme descrito na Seção 2.3, foi possível a aplicação de uma técnica de replicação tolerante a faltas de *crash* no nível do sistema anfitrião para reverter a limitação de SMIT não tolerar faltas de *crash* na máquina física (anfitrião), criando assim a abordagem DS-MIT (*Distributed* SMIT).

A estratégia de replicação utilizada foi a abordagem primário-*backup* [Budhiraja et al. 1993] em nível de anfitrião, que utiliza apenas uma máquina *backup*. Assim, as réplicas de serviço residentes na máquina primária formam o grupo de réplicas primário, que são ativas, recebendo e respondendo à requisições dos clientes. As réplicas de serviço residentes na máquina *backup* apenas executam as requisições recebidas através de sua *postbox* local, que é sincronizada à *postbox* primária por um processo no anfitrião. As réplicas residentes no anfitrião *backup* não recebem requisições diretamente dos clientes e tampouco enviam respostas a eles. A execução das requisições lidas da *postbox* pelo grupo de réplicas *backup* tem o único intuito de manter seu estado consistente com as réplicas de serviço de outras máquinas físicas.

No caso normal do protocolo, o cliente envia uma requisição somente para as réplicas do grupo primário. Tais réplicas escrevem propostas na *postbox*, seguindo o protocolo SMIT. Cada valor escrito é interceptado por um processo *daemon* executando

no sistema anfitrião. Antes de efetivar a escrita do valor na *postbox* local, o *daemon* envia o valor a ser escrito, através do canal privado, para o processo *daemon* residente na máquina *backup* e aguarda por confirmação de recebimento. Somente após o recebimento da confirmação é que a escrita do valor na *postbox* do primário é efetivada, garantindo que uma requisição somente seja efetivada nas VMs da primária quando já foi repassada às VMs *backup*. Caso o *daemon backup* não responda dentro de um determinado tempo, a primária o considera como faltoso e passa a ignorá-lo (lembrando que nosso modelo faz uma hipótese de sincronia no canal de comunicação entre os anfitriões). As réplicas de serviço residentes na máquina *backup* fazem a leitura de sua *postbox* local e executam as requisições ali propostas sem enviá-las ao cliente. Dessa forma, garantimos a consistência através das *postboxes* dos anfitriões, de modo que, se a máquina primária sofrer um *crash*, a máquina *backup* pode assumir o controle, mantendo o serviço correto e disponível.

4.1. Detecção de *Crash* do Primário

Para determinar quando a máquina primária sofre um *crash*, é necessário um mecanismo de detecção por parte da réplica *backup*. A implementação de tal mecanismo depende de premissas de sincronia no canal de comunicação que conecta as máquinas físicas.

O *daemon* da máquina primária envia periodicamente uma mensagem de notificação (vazia) para o daemon *backup*, de modo que este último possa saber que o primeiro permanece vivo. A cada τ segundos, a primária deve enviar uma mensagem para sua *backup*. Assim, o tempo entre duas mensagens de notificação é $\tau + \delta$, sendo δ a latência para que uma mensagem vazia trafegue de uma máquina para outra [Budhiraja et al. 1993]. Se uma mensagem de notificação não for recebida pelo *backup* após $\tau + \delta$ segundos, o *daemon backup* notifica suas máquinas virtuais locais para assumirem o posto das máquinas virtuais da máquina primária. Cada máquina virtual local ao *backup* então envia uma mensagem $\langle CHANGE - PRIMARY \rangle$ aos clientes, os quais então trocam o grupo de réplicas para o qual submetem suas requisições. Os clientes somente realizam tal mudança ao receberem $f+1$ mensagens $\langle CHANGE-PRIMARY \rangle$ assinadas e oriundas de diferentes réplicas pertencentes ao mesmo grupo. Isso evita réplicas maliciosas forçando a troca de grupo de réplicas por parte do cliente quando isso não é necessário. Uma visão geral da arquitetura distribuída é ilustrada na Figura 2.

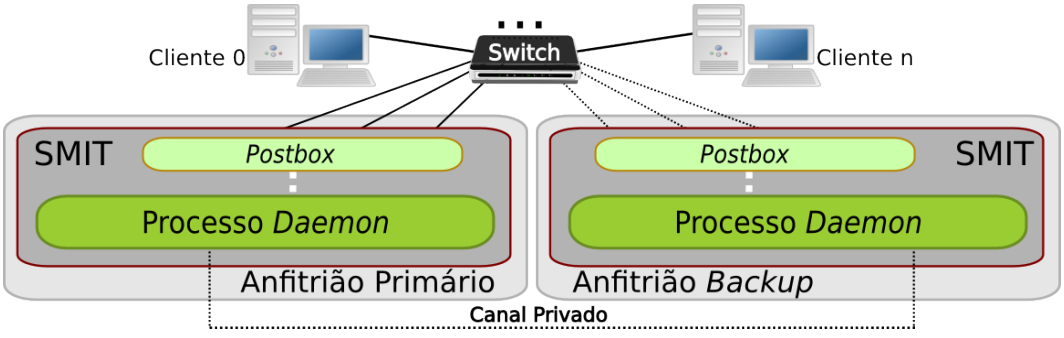

Figura 2. Arquitetura SMIT Distribuída (DSMIT)

5. Protótipo

Esta seção apresenta os experimentos realizados visando verificar a viabilidade prática do modelo e protocolos propostos nas seções anteriores. Dada a importância da *postbox* para

nosso modelo, a Subseção 5.1 fará uma discussão sobre alguns detalhes importantes na implementação de tal componente. Após isso, será realizada uma avaliação dos resultados obtidos com o protótipo implementado.

5.1. Implementação da *Postbox*

Como já fora descrito anteriormente, a *postbox* é o principal componente de nossa arquitetura. Para que a arquitetura como um todo se torne viável, é necessário que a *postbox* seja implementada de modo a cumprir as premissas e propriedades detalhadas na Seção 2.2.

Nossa implementação é baseada no VMM VirtualBox[2]. Tal escolha se deve ao fato de esse VMM oferecer a funcionalidade de compartilhamento de arquivos do sistema de arquivos do sistema anfitrião para os sistemas convidados (VMs). Além disso, o VirtualBox fornece um mecanismo de controle de acesso aos arquivos compartilhados, permitindo, via comandos emitidos pelo VMM, determinar quais máquinas virtuais possuem direito de escrita e leitura sobre quais arquivos compartilhados. Porém, o controle de acesso oferecido não é suficiente para que possamos garantir a propriedade *append-only*, que é crucial para a corretude da *postbox*. Para resolver tal problema, propusemos uma abordagem baseada no compartilhamento de múltiplos arquivos do anfitrião com os convidados, a qual chamamos de *n-write/n-read* (*nwnr*).

Seguindo essa abordagem, a *postbox* é implementada como um conjunto de recursos composto por um processo *daemon*, executado no sistema anfitrião, e dois conjuntos de arquivos, $RF = \{rf_0, rf_1, ..., rf_{N-1}\}$ e $WF = \{wf_0, wf_1, ..., wf_{N-1}\}$. Tanto RF quanto WF são conjuntos compostos por arquivos compartilhados pelo anfitrião para cada réplica (um arquivo para cada réplica). A diferença entre os dois conjuntos é que cada réplica (VM) utiliza seu arquivo do grupo RF para a leitura de valores da *postbox* e o seu arquivo do grupo WF para a escrita de valores na *postbox*. Assim, quando necessitar escrever um valor v na *postbox*, uma réplica de serviço s_i escreve tal valor no arquivo wf_i, o qual é compartilhado apenas entre o anfitrião e s_i. O papel do *daemon* é ler constantemente os arquivos do grupo WF e gravar atomicamente os valores lidos desses arquivos em todos arquivos do grupo RF, de modo que as réplicas de serviço tenham uma visão homogênea do conteúdo da *postbox*.

Através dessa abordagem, além da propriedade *append-only*, conseguimos prover maior isolamento entre as VMs para implementação da *postbox*. Tal isolamento beneficiou a simplicidade de implementação do protocolo de coleta de lixo sobre tal componente. Uma vez que uma VM possui seu par de arquivos de leitura e escrita isolados, a remoção de entradas desnecessárias passa a ser responsabilidade única de tal VM, sendo ela a única prejudicada por seu mau-comportamento. Para que isso seja verdadeiro, é necessária a implementação de um mecanismo de quotas de disco para os arquivos compartilhados, impedindo que estes cresçam indefinidamente.

5.2. Avaliação de Desempenho

A presente seção apresenta detalhes sobre o protótipo implementado do modelo e protocolo descritos nas seções anteriores. Para avaliar o desempenho do protótipo, utilizamos *micro-benchmarks*, metodologia de avaliação semelhante ao usado no PBFT

[2]Disponível em http://www.virtualbox.org/

[Castro and Liskov 2002]. Nesses *micro-benchmarks*, foram executadas operações nulas nas réplicas de serviço, variando os tamanhos da requisição e da resposta de 0 kB a 4 kB. A nomenclatura das operações se dá da seguinte forma: *ab*, onde *a* é o tamanho da requisição enviada e *b* é o tamanho da resposta para a requisição, ambos representados em kB. Por exemplo, uma operação chamada *04* possui uma mensagem de 0 kB como requisição e obtém como resposta uma mensagem de 4 kB.

O protocolo proposto neste trabalho foi implementado em Python, utilizando a versão 2.6.2 do interpretador. Para testá-lo, construímos o seguinte ambiente de testes:

- **Sistema Anfitrião Primário**: processador Core 2 Quad, com 8 GB de memória principal, executando o sistema operacional Debian GNU/Linux 5.0.
  - **VMM**: VirtualBox 2.2.4.
  - ***Postbox***: abordagem *n-write/n-read*.
  - **3 VMs**: cada uma com 1 GB de RAM, executando Ubuntu GNU/Linux 9.04 Server.
- **Anfitrião *Backup* (se aplica apenas ao cenário DSMIT)**: processador Core 2 Duo, com 1.5 GB de memória principal, executando Ubuntu 9.04 Desktop (demais detalhes são idênticos ao Anfitrião Primário, exceto que as VMs foram configuradas com 256 MB de memória apenas).
- **Clientes**: executando em um processador Core 2 Duo, com 2 GB de RAM, executando o sistema operacional Ubuntu GNU/Linux 9.04 Desktop.
- **Canal de Comunicação**: *switch* de rede de 100 Mb/s.
- **Mecanismo Criptográfico Padrão**: RSA, com chaves de 1024 bits, utilizando a ferramenta *Keyczar*[3]. A escolha por tal mecanismo é explicada na seqüência.

Avaliamos a execução de nosso protótipo através de medidas do tempo de resposta para certos tipos de operação. Para cada tipo de operação testada (variando o tamanho da requisição e da resposta), realizamos 10000 repetições, de forma que os valores apresentados na Figura 3 representam a média dos valores obtidos, representados em milisegundos. Inicialmente, criamos cinco cenários diferentes para a avaliação de nosso protótipo:

1. **SMIT**: o protótipo, com 3 VMs executando o protocolo proposto.
2. **SMIT-f**: mesmo que *SMIT*, porém com uma réplica faltosa.
3. **DSMIT**: a versão distribuída do protocolo, utilizando duas máquinas físicas com 3 VMs em cada uma destas.
4. **Single**: o serviço não-replicado.
5. **Single-VM**: o serviço não-replicado, executando sobre uma máquina virtual.

Como podemos ver na Figura 3, nosso protótipo gera um aumento substancial no tempo de resposta das operações. Tais resultados já eram esperados, visto que nosso protótipo oferece segurança a execução de uma requisição, necessitando as operações necessárias para a execução segura em um ambiente replicado, diferentemente dos ambientes não-replicados (*Single* e *Single-VM*). É importante perceber que o tamanho da resposta de uma operação possui efeito maior sobre o tempo de resposta do que o tamanho da requisição, pois o cliente precisa receber e votar sobre ao menos $f + 1$ respostas para completar a execução. Assim, operações que requerem grandes quantidades de dados como suas respostas levam a tempos de resposta maiores. Assim como no PBFT

[3]Disponível em http://www.keyczar.org/

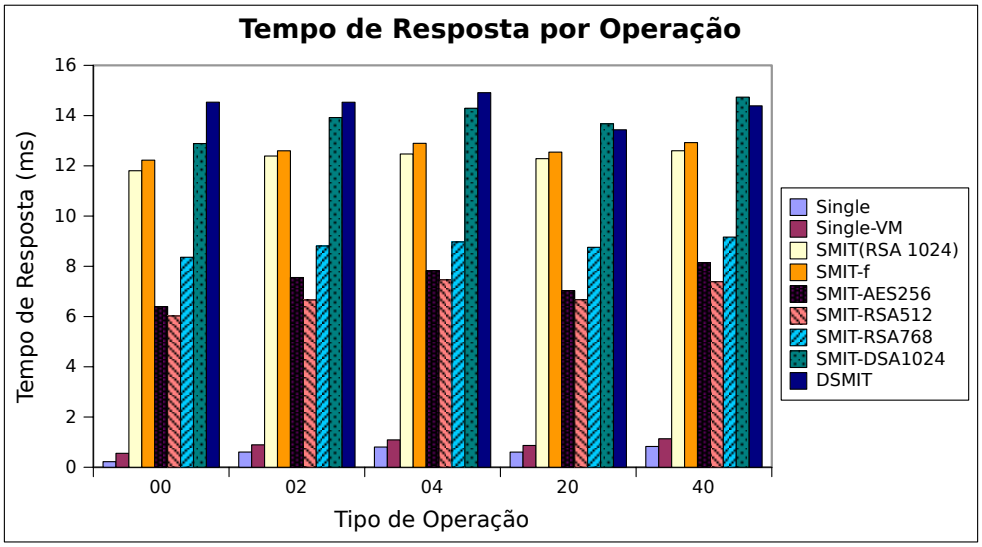

Figura 3. Tempo de Resposta por Operação

[Castro and Liskov 2002], isso pode ser otimizado fazendo com que a cada requisição enviada, o cliente eleja apenas uma das réplicas para lhe enviar a mensagem de resposta por inteiro, enquanto as outras lhe enviam apenas o resumo criptográfico destas. Caso os resumos recebidos não sejam idênticos ao resumo calculado sobre a mensagem inteira, o cliente solicita que todas as réplicas lhe enviem a mensagem inteira. Outro fato que merece atenção é o fato de o cenário *Single-VM* apresentar um aumento no tempo de resposta, se comparado ao caso mais simples (*Single*), mostrando que o uso de um ambiente virtualizado acarreta em um maior *overhead* para completar uma operação.

Avaliação dos Mecanismos Criptográficos Visando identificar o impacto causado pela utilização da técnica criptográfica utilizada em nossa implementação (RSA com chaves de 1024 *bits*) no tempo de resposta do protótipo, implementamos uma série de testes nos quais foi feita a alteração do mecanismo criptográfico utilizado no protótipo. Foram testados os algoritmos para criptografia assimétrica AES (com chaves de 256 *bits*), DSA (com chaves de 1024 *bits*) e RSA (com chaves de 512, 768 e 1024 *bits*). A Figura 3 apresenta também os resultados obtidos com a utilização dos diferentes algoritmos. De acordo com o gráfico, o protótipo obteve melhores resultados quando foram utilizados mecanismos criptográficos com chaves mais curtas. Os piores tempos de resposta ficaram por conta do algoritmo DSA com chaves de 1024 bits. Analisando os dados apresentados na Figura 3, optamos por manter o algoritmo criptográfico RSA com chaves de 1024 bits, visto que esse comprimento de chave oferece uma segurança maior que as alternativas que utilizam chaves mais curtas, mesmo perdendo em tempo de resposta.

Tempo de Recuperação de DSMIT Para avaliar o impacto que uma falta de *crash* no *host* primário gera para o cliente, foram realizadas 10 execuções do ambiente DSMIT, no qual o *host* primário sofre uma falta em meio a execução de requisições dos clientes. O tempo médio obtido para recuperação foi $225,95$ ms. Esse é o tempo necessário para que o anfitrião *backup* assuma o controle e responda as requisições pendentes. Tal valor foi

medido pelo cliente, através da diferença entre o tempo de resposta para a requisição cuja execução ocorreu em meio a falha do *host* primário e o tempo de resposta médio para execução de uma requisição no cenário sem falha do *host*.

6. Trabalhos Relacionados

A utilização do conceito e de tecnologias de virtualização para permitir que sistemas computacionais sejam tolerantes a faltas bizantinas tem sido alvo de pesquisas recentes na área de algoritmos distribuídos. A Arquitetura VM-FIT [Reiser and Kapitza 2007] é uma das primeiras propostas da qual se tem notícia na aplicação da virtualização para prover tolerância a faltas. A idéia básica desse trabalho é a execução de um serviço, de forma redundante, em várias máquinas virtuais sobre um único sistema físico. Nessa proposta, é necessária a existência de uma entidade confiável, que faz o papel ativo de coordenador do algoritmo, sendo responsável pela difusão das requisições dos clientes para as réplicas de serviço e pela votação sobre as respostas das réplicas.

O LBFT (*Lightweight Byzantine Fault Tolerance*) [Chun et al. 2008] é um trabalho cuja contribuição se concentra na discussão apenas teórica dos principais aspectos a serem considerados quando da construção de sistemas BFT sobre ambientes virtualizados, chamando atenção para pontos importantes e indicando caminhos a serem tomados.

Em ambos os trabalhos acima referidos, existe a presença de uma entidade confiável que é responsável por propor a ordem de execução das mensagens recebidas, bem como votar sobre as respostas geradas pelas réplicas. Em nossa proposta, quem propõe a ordem de execução das mensagens são as próprias réplicas de serviço, não sendo exigida a presença de uma VM exclusiva para tal tarefa.

Comparado ao PBFT [Castro and Liskov 2002], nossa proposta possui como vantagem a simplicidade com relação ao número de mensagens trocadas para efetivar uma requisição e também um menor número total de réplicas exigido para funcionamento do sistema (diferença de f réplicas).

7. Conclusão

Este artigo apresentou uma arquitetura e um algoritmo para replicação BFT utilizando virtualização para implementar a replicação de modo a oferecer um sistema tolerante a intrusões. Foi apresentada também uma abordagem baseada em um modelo híbrido, na qual é feita a replicação do sistema físico e das máquinas virtuais, sendo aplicado um modelo de falhas diferente para cada nível da replicação. A viabilidade prática da proposta foi verificada através de testes práticos sobre um protótipo desenvolvido, obtendo bons resultados. Os trabalhos futuros são focados na implementação da arquitetura utilizando outro VMM e em novas possibilidades de versões distribuídas.

Referências

Avizienis, A. and Kelly, J. P. J. (1984). Fault tolerance by design diversity: Concepts and experiments. *Computer*, 17(8):67–80.

Budhiraja, N., Marzullo, K., Schneider, F., and Toueg, S. (1993). The primary-backup approach. *Distributed systems*, 2:199–216.

Castro, M. and Liskov, B. (2002). Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461.

Chun, B. G., Maniatis, P., and Shenker, S. (2008). Diverse replication for single-machine byzantine-fault tolerance. *2008 USENIX Annual Technical Conference*.

Correia, M., Lung, L., Neves, N., and Veríssimo, P. (2002). Efficient Byzantine-resilient reliable multicast on a hybrid failure model. In *Proceedings of the Symposium on Reliable Distributed Systems*.

Correia, M., Neves, N., Lung, L., and Veríssimo, P. (2007). Worm-IT–a wormhole-based intrusion-tolerant group communication system. *The Journal of Systems & Software*.

Correia, M., Neves, N., and Verissimo, P. (2004). How to tolerate half less one Byzantine nodes in practical distributed systems. In *Proceedings of the Symposium on Reliable Distributed Systems*, pages 174–183.

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

Kotla, R., Alvisi, L., Dahlin, M., Clement, A., and Wong, E. (2007). Zyzzyva: speculative byzantine fault tolerance. In *Proceedings of ACM SIGOPS Symposium on Operating Systems Principles*.

Lamport, L., Shostak, R., and Pease, M. (1982). The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401.

Luiz, A. F., Bessani, A. N., Lung, L. C., and Filgueiras, T. (2008). Repeats-uma arquitetura para replicaçao tolerante a faltas bizantinas baseada em espaço de tuplas. In *Anais do XXVI Simpósio Brasileiro de Redes de Computadores*.

Nagappan, N., Ball, T., and Zeller, A. (2006). Mining Metrics to Predict Component Failures. In *Proceedings of the 28th International Conference on Software Engineering*.

Reiser, H. P. and Kapitza, R. (2007). VM-FIT: Supporting intrusion tolerance with virtualisation technology. In *Proceedings of the 1st Workshop on Recent Advances on Intrusion-Tolerant Systems*, pages 18–22.

Rivest, R., Shamir, A., and Adleman, L. (1978). A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):126.

Schneider, F. B. (1990). Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319.

Stumm Júnior, V., Lung, L. C., Correia, M., da Silva Fraga, J., and Lau, J. (2009). Desenvolvimento de Serviços Tolerantes a Intrusões Usando Máquinas Virtuais. In *Anais do XXXVI Seminário Integrado de Software e Hardware - SEMISH*.

Veronese, G., Correia, M., Bessani, A., and Lung, L. (2009). Spin One's Wheels? Byzantine Fault Tolerance with a Spinning Primary. In *28th IEEE International Symposium on Reliable Distributed Systems, 2009. SRDS'09*, pages 135–144.

Yin, J., Martin, J. P., Venkataramani, A., Alvisi, L., and Dahlin, M. (2003). Separating agreement from execution for byzantine fault tolerant services. *ACM SIGOPS Operating Systems Review*, 37(5):253–267.