Desenvolvimento de Serviços Tolerantes a Intrusões Usando Máquinas Virtuais

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

[1]Programa de Pós-graduação em Ciência da Computação – INE – Universidade Federal de Santa Catarina (UFSC) – Florianópolis – SC – Brasil

[2]LASIGE – Departamento de Informática – Faculdade de Ciências da Universidade de Lisboa – Portugal

[3]Programa de Pós-graduação em Engenharia de Automação e Sistemas – DAS – Universidade Federal de Santa Catarina (UFSC) – Florianópolis – SC – Brasil

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

Abstract. *Much research aiming to design practical algorithms to support Byzantine Fault-Tolerant distributed applications has been made in recent years. These solutions are designed to make the applications resistant to successful attacks against the system, thereby making services tolerant to intrusions. Recently, some of these studies have considered the use of virtual machines for building an environment of trusted computing. These proposals aim to implement service and operating system diversity at virtual machine level. This paper presents VMBFT, an architecture for Byzantine Fault-Tolerance using virtual machines.*

Resumo. *Diversas pesquisas para desenvolvimento de soluções práticas de suporte a aplicações distribuídas tolerantes a faltas bizantinas têm sido realizadas nos últimos anos. Estas soluções visam tornar as aplicações resistentes inclusive a ataques maliciosos bem sucedidos contra o sistema, tornando assim os serviços tolerantes a intrusões. Recentemente, alguns destes trabalhos têm considerado o uso de virtualização para a construção de um ambiente de computação confiável. Estas propostas visam implementar a diversidade de serviço e de sistema operacional em nível de máquinas virtuais. Este artigo apresenta VMBFT, uma arquitetura para tolerância a faltas bizantinas usando máquinas virtuais.*

1. Introdução

O papel que infraestruturas e sistemas computacionais desempenham em nossa sociedade vem crescendo em importância com o passar das últimas décadas. A dependência e a confiança depositadas sobre esses sistemas vêm aumentando consideravelmente, dia após dia. Dada a relevância desses sistemas para o funcionamento de serviços básicos para o nosso cotidiano, é necessário que estes se comportem corretamente mesmo sob a presença de faltas, as quais podem acarretar em grandes prejuízos, desde financeiros até humanos. Recentemente, as faltas em sistemas

computacionais têm aparecido mais frequentemente sob a forma de intrusões, que são o resultado de um ataque que obtém sucesso ao explorar uma ou mais vulnerabilidades [Correia 2005]. Para garantir que esses sistemas permaneçam disponíveis, corretos e seguros mesmo sob a presença de faltas e vulnerabilidades, é necessário que sejam desenvolvidos mecanismos para viabilizar a tolerância a intrusões nesses ambientes.

Visando garantir segurança para esses sistemas, o desenvolvimento de soluções para suporte a aplicações distribuídas tolerantes a faltas tem sido alvo de pesquisas há mais de vinte e cinco anos [Lamport, et al. 1982, Schneider 1990]. No entanto, somente nos últimos dez anos é que surgiram propostas com viabilidade prática para Replicação Máquina de Estados (ou replicação ativa) tolerante a faltas bizantinas (BFT) [Castro, et al. 1999, Castro, et al. 2002, Yin, et al. 2003, Correia, et al. 2004, Chun, et al. 2007, Kotla, et al. 2007, Luiz, et al. 2008, Veronese 2008].

Os protocolos para BFT, em geral, consistem na replicação de um determinado serviço em um conjunto de máquinas que se comunicam, objetivando prover um serviço seguro e confiável, mesmo sob a presença de um número limitado f de membros maliciosos (intrusos) que se comportem de forma diferente da especificação correta do protocolo. Esses protocolos visam possibilitar a implementação de serviços replicados capazes de atender aos requisitos de confiabilidade, integridade e disponibilidade, que são fundamentais para se alcançar a Confiança no Funcionamento (*Dependability* [Avizienis et. al. 2004]). Vários trabalhos com propostas para BFT têm apresentado soluções elegantes para, por exemplo, reduzir a complexidade de mensagens, reduzir o número de passos de comunicação para alcançar o acordo [Kotla, et al. 2007], e para otimizar o uso dos recursos computacionais disponíveis [Yin, et al. 2003, Correia, et al. 2004, Luiz, et al. 2008]. 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]). Para isso, estas soluções vão desde assumir premissas de sincronia mais relaxadas (porém, práticas e realistas) até a definição de arquiteturas híbridas com o uso de componentes seguros (os *wormholes* [Correia, et al. 2004, Correia, et al. 2007]).

Recentemente, surgiram alguns trabalhos propondo o uso de tecnologias de virtualização para implementação de sistemas tolerantes a intrusões em uma única máquina física [Bessani, et al. 2007, Reiser, et al. 2008]. Nessas propostas, cada réplica é executada em um ambiente virtualizado, sendo que todo o conjunto de réplicas reside no mesmo computador. Essas propostas permitem implementar o conceito de diversidade de serviços e de sistemas operacionais em nível de máquinas virtuais [Avizienis, et. al. 2004]. Ou seja, usar diversidade de projeto ao implementar cada réplica de um protocolo BFT em uma máquina virtual (VM – *virtual machine*) distinta (com sistema operacional distinto das outras VMs), com todas as VMs executando dentro de uma única máquina real. Uma vantagem óbvia dessa abordagem é a redução do custo de replicação para implementação de um serviço tolerante a intrusões pelo uso de uma única máquina física. Como contraponto, a máquina real é um ponto singular de falha: se a mesma sofre um *crash*, o serviço se torna indisponível. No entanto, para contornar tal situação, pode-se usar técnicas de replicação tradicionais, tolerantes a faltas de *crash*, para o sistema físico.

Vale ressaltar que apesar da ocorrência de falhas de máquinas se apresentarem mais frequentemente ligadas a faltas de *crash* do que a faltas bizantinas (maliciosas ou intrusões), os prejuízos são normalmente muito mais severos quando da ocorrência destas últimas faltas. Para exemplificar, podemos citar o caso de um intruso malicioso realizando operações financeiras (não autorizadas) de grande porte em servidores de um banco ou ainda, um intruso concretizando atos terroristas em um sistema computacional de uma usina nuclear. A literatura tem mostrado que, para tratar do problema dos ataques maliciosos sobre o software, as soluções devem adotar técnicas de replicação máquina de estado aliadas as técnicas de diversidade de projeto [Hiltunen, et al. 2003]. Essas propostas são fundamentadas na observação de que faltas bizantinas com intenção maliciosa (intrusão) ocorrem pela tentativa do uso das vulnerabilidades da porção do sistema computacional que é implementada em software (ou seja, sistema operacional, *drivers*, serviços, aplicações, etc.).

O presente trabalho se insere no desafio "**5 – Desenvolvimento tecnológico de qualidade: sistemas disponíveis, corretos, seguros, escaláveis, persistentes e ubíquos**", pois visa a busca de novas soluções tecnológicas, de baixo custo, em termos de modelos e arquiteturas, para suporte ao desenvolvimento de aplicações com alta disponibilidade, corretas e seguras mesmo em presença de ataques de segurança e intrusões [Lung, 2009]. Para isso, este trabalho propõe uma nova arquitetura para desenvolvimento de aplicações tolerantes a intrusões utilizando a tecnologia de virtualização e também propõe um algoritmo para execução de serviços replicados nessa arquitetura. O objetivo do trabalho é implementar a diversidade de réplicas de serviço em nível de máquinas virtuais, onde a comunicação entre as réplicas é feita através de uma abstração de memória compartilhada. Assim, não é necessária a comunicação direta, por passagem de mensagem via rede, entre os servidores replicados. Será mostrado, nesse modelo, que o número mínimo necessário de réplicas se reduz de $N \geq 3f+1$ para $N \geq 2f+1$, sendo f o número máximo de réplicas bizantinas, o que diminui o custo da replicação. Três trabalhos já haviam mostrado que essa redução é possível através da utilização de um componente/*wormhole* seguro [Correia, et al. 2004, Chun, et al. 2007, Veronese 2008]. O presente trabalho demonstra que esse componente seguro necessita apenas fornecer uma abstração de memória compartilhada simples para conseguir essa redução. Além do mais, diferentemente dos outros trabalhos, em nossa proposta a máquina hospedeira não tem função ativa no sistema [Reiser, et al. 2007, Reiser, et al. 2008], pois apenas fornece uma abstração para comunicação.

2. Modelo de Sistema

Em nosso modelo, além dos clientes externos ao ambiente virtualizado, temos duas categorias de sistemas: o **anfitrião**, que está rodando sobre o *hardware* físico, podendo ser representado pelo monitor de máquinas virtuais (VMM – *Virtual Machine Monitor*) ou por um sistema operacional com um VMM executando sobre si, e os **convidados**, sistemas virtuais rodando sobre o VMM. No sistema anfitrião reside a área de memória compartilhada que será usada para comunicação entre os sistemas convidados, os quais executarão os serviços replicados que implementam um protocolo BFT.

As premissas descritas a seguir são similares às adotadas pelos trabalhos relacionados (seção 6). Assumimos que um atacante pode dominar completamente as ações de f servidores replicados, porém o estrago que o mesmo pode causar ao sistema

fica restrito às f máquinas virtuais violadas. O confinamento é garantido no nosso modelo através de técnicas criptográficas usadas sobre as mensagens trocadas entre réplicas e clientes. Todas as mensagens enviadas no sistema contêm um código MAC (*Message Authentication Code* [Tsudik 1992]) de autenticação de mensagens, gerado a partir de chaves secretas compartilhadas entre os pares comunicantes. Assumimos também, em relação à técnica criptográfica citada acima, que um atacante não possui poder computacional para quebrá-la.

O uso desta técnica de criptografia simétrica determina a necessidade da geração de uma chave secreta para cada par <*cliente, réplica*>. Por exemplo, em um ambiente com 3 clientes e 5 réplicas, são necessárias 15 chaves secretas. O número de chaves poderia ser reduzido através da utilização de criptografia de chave pública, porém essa opção apresenta um elevado custo de desempenho [Castro, et al. 1999]. Os mecanismos criptográficos utilizados no modelo evitam também ataques do tipo *spoofing*[1]. Nosso modelo assume que o sistema operacional anfitrião pode apresentar vulnerabilidades, porém estas não podem ser exploradas pelos sistemas virtuais. Para que essa premissa seja verdadeira, confiamos que o VMM forneça o isolamento necessário para a execução segura dos sistemas virtuais. Além disso, o nosso modelo necessita, por construção, que o sistema anfitrião seja externamente inacessível. Isso pode ser obtido através de técnicas de *firewall*, bloqueando o acesso à interface de rede do sistema anfitrião. Apesar de o sistema anfitrião ser inacessível, o modelo mantém as interfaces dos sistemas convidados (máquinas virtuais) visíveis através da rede, de forma que estes possam ser acessados pelos clientes.

Neste trabalho, assumimos um sistema assíncrono, composto por servidores e clientes. Os clientes são conectados aos servidores através de um canal de comunicação ponto a ponto confiável. O conjunto de servidores, $S = \{S_0, S_1, ..., S_{n-1}\}$, também chamados de réplicas de serviço, é um conjunto mínimo de n sistemas virtuais executando sobre um único sistema físico. As réplicas de serviço se comunicam entre si através de uma área de memória de acesso comum a todas, não necessitando, portanto, de acesso à rede para comunicação entre si. As réplicas de serviço podem desempenhar dois tipos de papéis no sistema: (i) a réplica primária que é a responsável por determinar a ordem na qual as requisições são processadas, e (ii) as réplicas *backup* que são os sistemas virtuais restantes, que seguem a ordem proposta pela primária para a execução das requisições. Os clientes, representados pelo conjunto $C = \{C_0, C_1, ...\}$ são sistemas que se comunicam com as réplicas de serviço através do envio de mensagens pela rede. No nosso modelo, são toleradas até f réplicas faltosas (faltas bizantinas [Lamport, et al. 1982]) de um mínimo de $2f+1$ processos. O comportamento bizantino envolve, no nosso modelo, a parada de execuções, a omissão no envio ou recepção de mensagens e envio de mensagens inconsistentes. Os clientes, em nosso modelo, devem apresentar comportamento correto.

[1]Ataque no qual é feita uma falsificação sobre o remetente de uma mensagem.

Técnicas de diversidade de *software*[2] [Avizienis, et al. 2004] são aplicadas na construção das réplicas de serviço. A idéia é que as réplicas falhem de forma independente, ou seja, a falha de uma réplica não implica na falha de outras. Essa diversidade pode ser implementada em nível de sistema operacional e de aplicação (linguagem de programação e metodologias de desenvolvimento). Nosso modelo não tolera faltas de *crash* (parada) na máquina física, associadas ao sistema anfitrião, ao VMM e ao suporte físico. Uma falta de *crash* ocorrida no sistema anfitrião implica diretamente na parada dos sistemas virtuais. A utilização de técnicas tolerantes a faltas de *crash*, como replicação ativa da máquina física, pode ser aplicada ao nosso modelo de forma a torná-lo também tolerante a esse tipo de faltas.

Conforme descrito anteriormente, nosso algoritmo faz uso de uma região de memória compartilhada, que será também chamada de *caixa postal*, através da qual as réplicas se comunicam. Esta abstração de memória compartilhada deve ser oferecida pelo sistema anfitrião para os sistemas convidados. Qualquer réplica grava na região de memória um valor para ser lido pelas outras réplicas. Todas as réplicas lêem os valores na mesma ordem em que estes foram escritos. A *caixa postal* possui uma interface básica de acesso, composta por dois métodos:

```
append(valor): boolean
read(): valor
```

O método ***append*** grava um valor na *caixa postal* juntamente com o identificador da réplica que escreveu (não pode haver falsificação nesses últimos dados, pois a *caixa postal* é manipulada através da VMM). O método retorna um valor do tipo *boolean* indicando se a escrita ocorreu com sucesso ou não. O ritmo a que cada réplica pode escrever na *caixa postal* é limitado para impedir ataques de negação de serviço. O método ***read***, por sua vez, retorna um valor anteriormente escrito. Essas leituras retornam sempre o valor seguinte ao da última leitura.

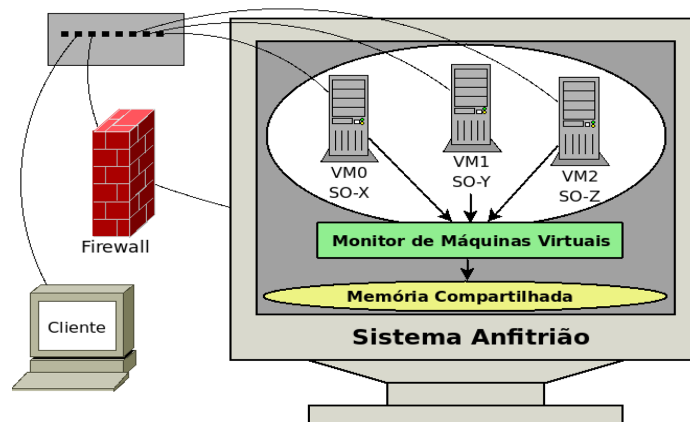


Figura 1. Visão Geral da Arquitetura

[2] Técnicas de diversidade de software são apropriadas para evitar a presença de vulnerabilidades comuns nas diversas réplicas de serviço. Cada réplica executando um software diferente, mas com a mesma funcionalidade, minimiza a ocorrência das mesmas vulnerabilidades no conjunto total de réplicas.

Uma vez escrito na *caixa postal*, um valor não pode ser alterado, pois esta opera em modo *append-only*. Isso evita que uma réplica maliciosa forneça valores inconsistentes para serem vistos, em tempos diferentes, pelas demais réplicas. Sendo a *caixa postal* um componente único e centralizado na arquitetura, os conjuntos de leituras realizadas por todas as réplicas possuem sempre a mesma ordem. Os controles de acesso e de concorrência sobre a memória compartilhada são de responsabilidade do VMM. É importante perceber que a *caixa postal* se trata de um componente de capacidade finita. Para evitar que o excesso de escritas ultrapasse a capacidade da memória, causando a parada do protocolo, é necessário um mecanismo de limpeza de entradas da *caixa postal* que já tiveram sua requisição correspondente processada pelas réplicas corretas. A figura 1 ilustra a visão geral da arquitetura, mostrando a estrutura interna do sistema anfitrião.

3. Propriedades do Algoritmo

O algoritmo proposto neste trabalho segue a linha dos algoritmos de Replicação de Máquinas de Estado [Schneider 1990]. De maneira geral, algoritmos distribuídos são desenvolvidos de forma a satisfazer as seguintes propriedades:

- Segurança (*Safety*): um serviço distribuído deve satisfazer a linearizabilidade, ou seja, o serviço replicado deve se comportar da mesma forma como se fosse implementado de forma centralizada [Castro and Liskov 1999];
- Vivacidade (*Liveness*): um cliente que faz uma requisição ao sistema distribuído, em algum momento deverá receber a resposta à sua requisição.

Para garantir que as propriedades acima sejam satisfeitas, nosso algoritmo deve ser executado em um ambiente onde no máximo $(N-1)/2$ réplicas apresentem comportamento incorreto, ou seja, o sistema necessita de um total de $N \geq 2f+1$ réplicas.

No modelo Máquina de Estado, é necessário que as requisições emitidas pelos clientes sejam todas recebidas (acordo) e executadas na mesma ordem (ordem total) pelas réplicas do serviço. As réplicas corretas apresentam comportamento determinista, ou seja, todas as réplicas corretas iniciam sua execução no estado inicial e terminam no mesmo estado final.

Para que o conjunto de réplicas execute as mesmas requisições e na mesma ordem, é necessário um protocolo de consenso ou seu equivalente, um protocolo de difusão atômica. No PBFT [Castro and Liskov 1999] que é um protocolo de suporte a replicação Máquina de Estado, os aspectos de acordo e ordem total são definidos através de um líder envolvendo trocas de mensagens, ou seja, o líder dissemina para as outras réplicas sua proposta de ordem, através da escolha de uma das requisições do cliente. Na nossa proposição, a ordem total é definida de forma bastante simples pela réplica primária, que grava na *caixa postal* os resumos criptográficos das mensagens recebidas dos clientes, na sua ordem de recebimento. O acordo é conseguido com a comunicação direta do cliente com as diversas réplicas e pelos mecanismos de verificação (*hashes*) calculados e escritos na *caixa postal* pelo líder.

No nosso modelo, por se tratar de uma região de memória que é vista de forma homogênea pelas réplicas, é possível evitar todos os passos de comunicação exigidos nos algoritmos de acordo anteriores. Uma vez que a réplica primária tenha escrito sua

proposta de ordem para uma requisição na *caixa postal*, basta que as outras réplicas leiam esse valor e executem a requisição correspondente.

4. Algoritmo

O algoritmo segue uma sucessão de configurações chamadas de *visões*. Em cada visão ***v***, a réplica ***s*** é a réplica primária sse $s = v \bmod |S|$. As réplicas restantes são *backups*. Se o primário for faltoso, pode ser necessário fazer uma mudança de visão (seção 4.1).

Em uma breve descrição, o modo normal de funcionamento do algoritmo é composto pela seguinte seqüência de fases:

1. O cliente ***c*** envia uma requisição de serviço para todas as réplicas;
2. O primário determina a ordem da requisição dentro do conjunto de requisições e comunica esse valor aos *backups*, através da escrita do resumo criptográfico desta na *caixa postal*;
3. Todas as réplicas executam a requisição na ordem proposta pelo primário e enviam a resposta diretamente ao cliente;
4. O cliente ***c*** verifica a maioria de mensagens iguais para determinar a resposta correta.

No modo normal de funcionamento, apenas o primário deve escrever valores na *caixa postal* e esses valores obedecem ao formato *<ORDER-REQ, id_cliente, timestamp, hash(req)>*. Se outra réplica escrever valores na *caixa postal*, eles são simplesmente ignorados pelas réplicas corretas.

Para descrever o algoritmo de forma mais detalhada, detalharemos a sequência de passos envolvida para concretizar a execução de uma requisição feita por parte do cliente. Essa descrição será acompanhada por uma análise dos algoritmos 1 e 2. A figura 2 ilustra a seqüência de passos que será descrita a seguir.

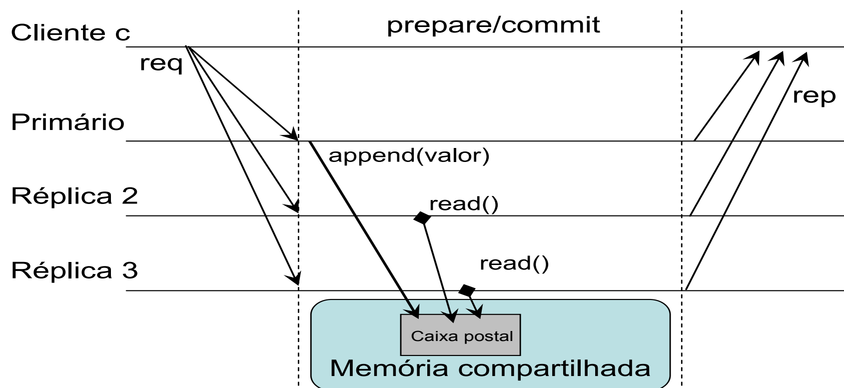


Figura 2. Diagrama de Mensagens do VMBFT

Algoritmo 1. Envio de requisições por parte de um Cliente C

send_request():
1: **send** *req* **to** *all replicas* **in** *S*
2: **wait until** *f+1 matching replies* **from** *S*

Algoritmo 2. Ordenação e execução das requisições no Conjunto de Réplicas S

```
receive():
  1:  receive req from client
  2:  if primary then
  3:    write prop(hash(req)) to postbox
  4:  add req to buffer

process_request():          {tarefa chamada enquanto houverem mensagens no buffer}
  1:  read prop from postbox V timeout
  2:  if prop = Ø then                                        {expirou o timeout}
  3:    write <VIEW-CHANGE,i,view+1> to postbox
  4:    view_change()
  5:  else if prop is VIEW-CHANGE then
  6:     add prop to buffer
  7:     view_change()
  8:  else                                                    {prop é um hash}
  9:    req ← search for prop in buffer V timeout
 10:    if req = Ø then                                       {expirou o timeout}
 11:      write <VIEW-CHANGE,i,view+1> to postbox
 12:      view_change()
 13:    else
 14:      execute req and send reply

view_change():
  1:  if f+1 matching VIEW-CHANGE in buffer then
  2:    view = view + 1
  3:    process each remaining request from view-1
```

***Fase 1**. Cliente **c** envia requisição para todas as réplicas.*

O cliente ***c*** envia uma solicitação de execução de uma operação **op** para todas as réplicas. A mensagem enviada possui a forma *<REQUEST, c, t, op>*, onde ***t*** é o *timestamp* do momento em que o cliente solicita a operação. Esse *timestamp* é utilizado pelas réplicas para a garantia da semântica de execução *exactly-once*. Esta fase é representada pela linha 1 do algoritmo 1.

***Fase 2.a**. Réplica **primária** recebe a requisição enviada pelo cliente, determina a ordem desta no conjunto de requisições e grava na caixa postal o resumo da mensagem recebida do cliente.*

A réplica primária na visão ***v*** recebe a requisição *<REQUEST, c, t, op>*, verifica se o valor de ***t*** é maior do que o valor correspondente das requisições anteriores do mesmo cliente e valida a assinatura da mensagem para garantir a integridade e a autenticidade desta.

O valor do *timestamp* ***t*,** por ser sempre maior do que os valores anteriores, implica em que as requisições vindas de um mesmo cliente sejam totalmente ordenadas. O fato das réplicas guardarem o *timestamp* dos pedidos de cada cliente permite a garantia da semântica *exactly-once*.

A assinatura da mensagem é gerada utilizando a *chave secreta* compartilhada entre o cliente e a réplica durante a fase de inicialização do algoritmo. Dessa forma, é possível verificar a autenticidade e a integridade da mensagem.

Conforme mostra o passo *prepare/commit* da figura 2, a réplica primária grava na *caixa postal* uma estrutura da forma *<ORDER-REQ, c, t, h>*, onde ***h*** é o *hash*

computado sobre a mensagem recebida do cliente. Conforme descrito na seção 2, todas as réplicas lêem as escritas na *caixa postal* pela mesma ordem, então, ao escrever o *hash* da mensagem nesta, o primário está determinando a ordem na qual essa mensagem deverá ser processada pelos *backups*. Assim, todas as réplicas irão processar as mensagens na ordem em que os resumos correspondentes foram escritos pela primária na *caixa postal*, o que garante a ordem total de execução das requisições pelas réplicas.

Essa fase pode ser vista no algoritmo 2, tarefa *receive()*, nas linhas de 1-3.

Fase 2.b*. Réplicas backup recebem a requisição enviada pelo cliente e consultam a caixa postal visando encontrar o resumo desta requisição escrito pelo primário.*

Após receber uma mensagem <*REQUEST, c, t, op*> enviada pelo cliente, a réplica passa a consultar periodicamente a *caixa postal* em busca de uma entrada <*ORDER-REQ, c, t, h*> que contenha no campo *h* um valor idêntico ao *hash* da mensagem recebida (fase *prepare/commit* da figura 2). A ordem na qual essa entrada for encontrada na *caixa postal* determina a ordem em que a requisição correspondente, já armazenada em *buffer*, deve ser executada pela réplica. Caso uma entrada não seja encontrada após um período de tempo pré-definido (linhas 2-4 da tarefa *process_request()*) ou caso a entrada lida da *caixa postal* não corresponda à requisição alguma recebida pela réplica após uma espera de tempo pré-definido (linhas 10-12 da tarefa *process_request()*), a réplica *backup* desconfia da primária e propõe uma mudança de visão.

Essa fase pode ser vista no algoritmo 2, nas linhas de 1-4 da tarefa *receive()* (recepção da mensagem) e nas linhas 1-12 da tarefa *process_request()* do algoritmo 2 (checagem na *caixa postal* em busca da ordem para a mensagem).

Fase 3*. Cada réplica executa a requisição na ordem especificada pelo primário e envia a resposta diretamente para o cliente.*

Quando chegar o momento apropriado para execução de uma requisição, cada réplica, tanto a primária quanto as *backups*, irá executar a requisição. Após essa execução, as réplicas enviam o resultado desta, devidamente assinado, diretamente para o cliente em uma mensagem da forma <*REPLY, c, v, t, data*>, onde *data* representa o conteúdo em si da resposta a ser enviada. Além disso, a réplica deve armazenar em uma variável local o *timestamp* da mensagem processada.

A fase 3 é representada na linha 14 da tarefa *process_request()* no algoritmo 2.

Fase 4*. Cliente **c** aguarda por f+1 mensagens iguais, vindas de diferentes réplicas.*

Após enviar a requisição para as réplicas, o cliente aguarda por *f+1* respostas da forma <*REPLY, c, v, t, data*> vindas das réplicas. Caso todas as mensagens deste conjunto sejam iguais – com relação ao conteúdo da resposta (campo *data*), ao número da visão ***v***, ao *timestamp* ***t*** e ao destinatário ***c*** – e devidamente assinadas pelos seus remetentes, a requisição é considerada correta e o cliente pode prosseguir sua execução. Caso contrário, o cliente aguarda por respostas suficientes até obter *f+1* mensagens iguais no conjunto de respostas recebidas. Após isso, o cliente segue sua execução. Esta fase é representada pela linha 2 do algoritmo 2.

Um detalhe importante sobre o algoritmo 2, o qual não está incluído na descrição do mesmo, é que para evitar que um primário malicioso postergue indefinidamente a

proposição de ordem para uma requisição recebida do cliente, as réplicas *backup* iniciam um temporizador ao receber uma requisição, caso não haja um temporizador ativo para outra mensagem. Ao receber a ordem para execução dessa mensagem através da *caixa postal*, a réplica pára o temporizador. Caso volte a esperar por uma ordem para alguma outra mensagem, a réplica reinicia o temporizador e, caso este expire, a réplica propõe uma mudança de visão. Este mecanismo também é utilizado no PBFT [Castro, et al. 1999].

4.1. Protocolo de Mudança de Visão

Quando a réplica primária falha, é necessário que as *backups* detectem essa falha e iniciem uma mudança de visão (troca de primária). Em nosso algoritmo, as réplicas *backups* suspeitam da réplica primária em dois casos:

a) *Backups* recebem uma requisição do cliente e, após um *timeout* pré-definido, ainda não obtiveram da *caixa postal* a ordenação correspondente, ou seja, não encontraram o resumo dessa requisição em uma entrada da *caixa postal*;

b) *Backups* lêem uma entrada da caixa postal e, após um *timeout* pré-definido, ainda não obtiveram a mensagem correspondente do cliente.

Quando uma réplica ***s***, correta, lê *f+1* mensagens *VIEW-CHANGE* (possivelmente incluindo a sua própria), esta tem a certeza de que pelo menos uma réplica correta está suspeitando do primário (pois de *f+1* pelo menos uma é correta) e realiza a seguinte seqüência de passos (representada pela tarefa *view_change()* no algoritmo 1):

1. Incrementa ***v***, onde ***v*** é o número da visão atual;
2. A réplica com número de identificação igual a $s' = v \mod |S|$ é identificada como a nova primária. Neste caso, duas situações podem ocorrer:
   2.1. Se $s = s'$, a réplica começa a atuar como primária;
   2.2. Se $s \neq s'$, a réplica começa a aceitar apenas valores *ORDER-REQ* escritos por *s'*.
3. O protocolo segue sua execução normal.

Como todas as réplicas corretas lêem a mesma seqüência de valores da *caixa postal,* todas seguem a mesma seqüência de visões. Convém notar que a réplica primária pode ser identificada como maliciosa e a visão mudar mesmo que isso não seja verdadeiro, simplesmente porque o sistema está lento. Isso é o mesmo que acontece com outros protocolos BFT e não adiciona qualquer problema na consistência do sistema.

Quando há uma mudança de visão, o *timeout* é duplicado para garantir que o algoritmo faz progresso no caso de os atrasos de comunicação serem muito altos (lembre que fizemos a hipótese de que esses tempos não crescem para sempre).

5. Implementação Protótipo

Implementamos um protótipo que consiste de um serviço NFS replicado através de máquinas virtuais dentro de uma única máquina física, a fim de verificar a viabilidade

prática do modelo e do algoritmo proposto no presente trabalho. O protótipo foi implementado em linguagem Java, utilizando como serviço uma implementação aberta do NFS (*Network File System*), o JNFS[3].

Para a implementação da *caixa postal* desta primeira versão do protótipo foi utilizada uma funcionalidade provida por alguns VMMs (p. ex. *VirtualBox*, *VMWare*), a qual permite que um diretório do sistema de arquivos do anfitrião seja compartilhado com os sistemas convidados. Assim, foi possível utilizar um arquivo pertencente a esse diretório para representar a abstração de memória compartilhada necessária em nosso modelo. As máquinas virtuais foram devidamente configuradas para que obtenham acesso a este diretório.

5.1. Avaliação de Desempenho

O ambiente de testes para avaliação de desempenho é composto por duas máquinas. Uma delas, o servidor, é responsável por "hospedar" as réplicas virtuais do serviço NFS e a outra, conectada à primeira através da rede, executa o cliente do serviço. O servidor é equipado com processador *Core 2 Quad*, com 8 GB de memória principal, interface de rede *Gigabit,* com sistema operacional *Ubuntu GNU/Linux 8.10, kernel* versão *2.6.27-7*. A máquina cliente é equipada com processador *Core 2 Duo*, com 2 GB de memória principal, interface de rede *Gigabit,* executando *Ubuntu GNU/Linux 8.10, kernel* versão *2.6.27-7*.

Os sistemas convidados, que executam o serviço NFS replicado, são máquinas virtuais com 1 GB de memória principal, executando sobre o VMM *Sun xVM VirtualBox* 2.0.4. Nos testes foram usadas três máquinas virtuais, nas quais foram instalados os seguintes sistemas operacionais: *Debian GNU/Linux 4.0*, *Ubuntu GNU/Linux Server 8.04* e *Ubuntu GNU/Linux Desktop 8.04*. Tanto nos sistemas convidados quanto no sistema cliente, foi instalada a máquina virtual Java, versão 1.6, necessária para a execução do protótipo implementado.

Para a realização dos experimentos, foram analisadas as seguintes operações sobre o sistema NFS replicado: (*i*) escrita de arquivos remotos (figura 3(a)) e (*ii*) leitura de arquivos remotos (figura 3(b)). Ambas as operações foram realizadas sobre arquivos de 4 kB, 8 kB, 16 kB e 32 kB. Para cada um dos casos foram realizadas 500 execuções, sendo feitas medidas do tempo de resposta de cada execução. Através desses valores, foi calculado o tempo de resposta médio para cada um dos tipos de operações anteriormente descritas. A fim de obter dados para comparação, foi realizada a avaliação em quatro cenários distintos:

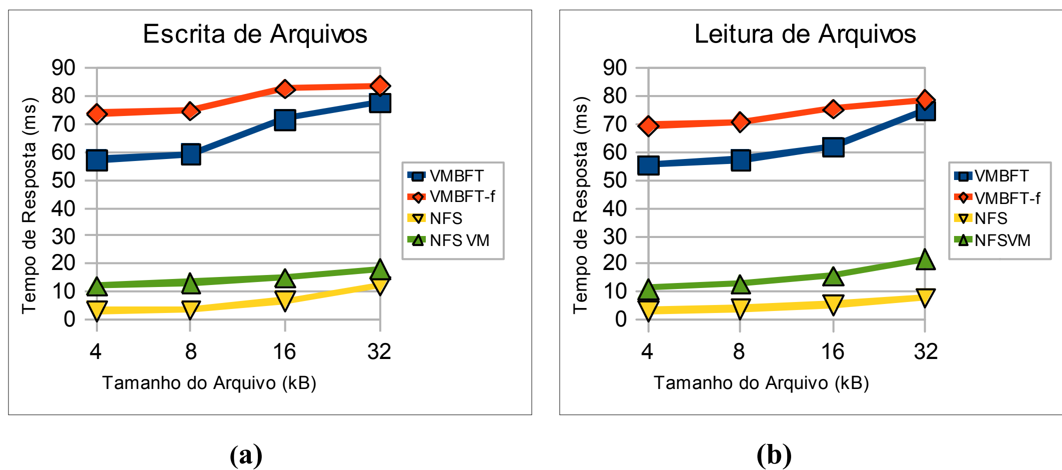
1. **VMBFT**: o protótipo implementado, executado apenas com réplicas corretas;
2. **VMBFT-f**: o protótipo implementado, executado com a presença de uma réplica faltosa;
3. **NFS**: sistema operacional executando diretamente sobre o hardware, com um servidor NFS respondendo a requisições;

[3] Disponível em http://www.void.org/~steven/jnfs/

4. **NFS VM**: sistema operacional virtual executando sobre um VMM, com um servidor NFS respondendo as requisições;

Por se tratar de uma arquitetura que adiciona segurança à execução de um serviço NFS, é natural que os cenários 1 e 2 apresentem um aumento no tempo médio de resposta das requisições, se comparado à execução singular do NFS (cenário 3). Analisando a figura 3, percebemos que os tempos de execução do NFS não replicado e não virtualizado são baixos se comparados aos resultados dos outros cenários. Além disso, percebemos o grande aumento no tempo médio de resposta que foi gerado pela execução do serviço NFS em uma única máquina virtual, sem replicação (cenário 4). Esse cenário de testes foi incluído na bateria de testes para que pudéssemos verificar na prática qual o impacto causado pela execução do serviço em um ambiente virtualizado. Dessa forma, o aumento no tempo de resposta gerado pela execução do serviço replicado sobre o protótipo VMBFT está dentro do esperado, visto que são três máquinas virtuais concorrendo por recursos da máquina física, principalmente pelo disco, devido em parte à comunicação via *caixa postal* e, também, à execução do serviço NFS, o qual faz uso intensivo do disco.

Além do caso otimista (cenário 1), onde todas as réplicas se comportam corretamente, testamos o protótipo em um cenário no qual uma réplica se comporta de forma bizantina, enviando respostas incorretas ao cliente (cenário 2). Conforme a figura 3 ilustra, o tempo médio de resposta nesse caso cresceu. Isso se deve ao fato de, em algumas execuções, o cliente receber respostas divergentes entre duas réplicas (uma correta e outra incorreta) e então aguardar por mais uma resposta para poder determinar qual delas é a correta. Essa situação não ocorre na execução em um ambiente sem faltas, pois a réplica aguarda apenas por ***f+1*** respostas iguais (o que no cenário 1 sempre ocorre com as primeiras ***f+1*** respostas) para considerar a resposta como correta.



(a) **(b)**

Figura 3. Comparação entre operações de escrita (a) e leitura (b) de arquivos em quatro cenários distintos

Em [Camargos, et al. 2008], foram feitas análises de desempenho sobre os principais VMMs disponíveis no mercado, dentre os quais o *VirtualBox*, que foi usado para implementação deste protótipo. Em uma das avaliações realizadas, que diz respeito a uma tarefa que faz uso intensivo da CPU e do disco, o *VirtualBox* apresentou um dos piores desempenhos dentre as soluções testadas, ficando muito abaixo de outros VMMs.

Assim, pode-se justificar em parte a queda no desempenho obtida quando da execução do serviço NFS em uma única máquina virtual e replicado no protótipo implementado.

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 por trás desse trabalho é a execução de um serviço, de forma redundante, em várias máquinas virtuais sobre um único sistema físico, o que permite diminuir o custo de sistemas tolerantes a faltas que, tradicionalmente, utilizam replicação física de *hardware* para criar a redundância de um determinado serviço. Nessa proposta, é necessária a existência de uma entidade confiável, que faz o papel de coordenador do algoritmo, pois este é responsável por fazer a difusão das requisições dos clientes para as réplicas de serviço e pela votação sobre as respostas das réplicas. O comprometimento desse domínio causaria o comprometimento de todo o protocolo de execução.

Outros algoritmos foram propostos, aplicando a idéia de replicação de sistemas em um sistema físico único. Os algoritmos LBFT1 e LBFT2 [Chun, et al. 2008] propõem uma abordagem na qual um sistema virtual confiável faz o papel de coordenador do algoritmo, sendo responsável, portanto, pela determinação da ordem na execução de requisições por parte das réplicas de serviço. Nesse trabalho, o coordenador é uma máquina virtual diferenciada dos outros, pois não é uma réplica de serviço, sendo responsável apenas por executar tarefas de coordenação. Assim, ambos os algoritmos propostos nesse trabalho toleram f faltas utilizando um mínimo de $2f+2$ VMs, sendo que uma destas é o coordenador confiável e as restantes $2f+1$ são as réplicas do serviço.

Tabela 1. Comparativo entre os Algoritmos Tolerantes a Faltas Bizantinas

	PBFT	VM-FIT	LBFT1	LBFT2	VMBFT
Número Mínimo de Réplicas	$3f+1$	$C+2f+1$	$C+2f+1$	$C+2f+1$	$2f+1$
Passos de Comunicação das Réplicas	3	2	1	2	1

A tabela 1 faz um comparativo entre os algoritmos acima citados (*C* significa coordenador), o PBFT [Castro and Liskov 1999] e o algoritmo proposto no presente trabalho, o VMBFT, com relação ao número mínimo de réplicas exigido pelo protocolo e ao número de passos de comunicação, entre réplicas, necessários para que estas concretizem uma requisição. A diferença do VMBFT em relação a esses sistemas é a inexistência de um coordenador confiável. No VMBFT, o VMM apenas provê uma abstração de comunicação segura.

Em alguns sistemas de replicação ativa, o tempo de comunicação entre réplicas não cresce indefinidamente, mesmo considerando sistemas assíncronos. Esta hipótese é feita, por exemplo, no protocolo PBFT original e outros protocolos de BFT na literatura [Castro and Liskov 1999, Castro and Liskov 2002, Yin, et al. 2003, Correia, et al. 2004, Chun, et al. 2007, Kotla, et al. 2007, Luiz, et al. 2008, Veronese 2008]. A hipótese é necessária para contornar o resultado FLP [Fischer, et al. 1985]. No nosso caso, essa

hipótese é menos impactante, pois envolve somente as comunicações entre clientes e réplicas. As outras comunicações do modelo são feitas usando memória compartilhada.

7. Conclusões

Este artigo apresentou uma arquitetura e um algoritmo para replicação tolerante a faltas bizantinas que utiliza o conceito de virtualização para implementar replicação e diversidade de serviços de forma a prover como resultado final um sistema tolerante a intrusões menos custoso que soluções anteriormente propostas. Para verificar a viabilidade do modelo proposto, foi desenvolvido e avaliado um protótipo, no qual foram utilizadas três máquinas virtuais replicando um servidor NFS. Através dessa implementação, verificamos que o modelo é viável na prática.

Uma discussão importante deve ser feita sobre o trabalho proposto. O fato de a arquitetura ser baseada em um único *host* físico hospedando todas as réplicas de serviço deixa o modelo vulnerável à faltas de *crash*. Uma vez que o sistema físico sofra um colapso, todas as réplicas de serviço serão afetadas. Para contornar tal situação, pode-se utilizar técnicas de replicação tradicionais para replicar fisicamente a máquina anfitriã do VMBFT, tornando assim, o sistema tolerante a faltas de *crash*.

Os trabalhos futuros se concentram no aprimoramento da arquitetura e do algoritmo para que suportem a presença de clientes maliciosos, na implementação da arquitetura proposta utilizando outros VMMs (p. ex. *KVM*, *Xen*, *User Mode Linux*), na possibilidade de criação de uma *caixa postal* em nível de memória principal e aumento da diversidade de software.

Agradecimentos

Os autores agradecem ao CNPq pelo apoio financeiro realizado através dos projetos 472754/2008-4 e 305430/2007-6.

Referências

Avizienis, A., Laprie, J.-C., Randell, B., and Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. IEEE Transactions on Dependable and Secure Computing, 1(1):11–33.

Bessani, A. N., Sousa, P., Correia, M., Neves, N. F. and Veríssimo, P. (2007). Intrusion-Tolerant Protection for Critical Infrastructures. DI/FCUL. Tech. Report 07-8, 2007.

Camargos, F. L., Girard, G. and Ligneris, B. d. (2008). Virtualization of Linux Servers: a comparative study. Linux Symposium, Ottawa, Canada.

Castro, M. and Liskov, B. (1999). Practical Byzantine Fault Tolerance. Proceedings of the third Symposium on Operating Systems Design and Implementation. New Orleans, Louisiana, United States, USENIX Association.

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., Maniatis, P. and Shenker, S. (2008). Diverse Replication for Single-Machine Byzantine-Fault Tolerance. USENIX Annual Technical Conference.

Chun, B., Maniatis, P., Shenker, S. and Kubiatowicz, J. (2007). Attested Append-Only Memory: Making Adversaries Stick to their Word. Proceedings of 21st ACM Symposium on Operating Systems Principles, ACM Press New York, NY, USA.

Correia, Miguel P. (2005) Serviços Distribuídos Tolerantes a Intrusões: resultados recentes e problemas abertos. V Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais - Livro Texto de Mini-cursos, pp. 113-162.

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 80(2):178-197.

Correia, M., Neves, N. and Verissimo, P. (2004). How to tolerate half less one Byzantine nodes in practical distributed systems. Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems.

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

Hiltunen, M., Schlichting, R. and Ugarte, C. (2003). "Building Survivable Services Using Redundancy and Adaptation." IEEE Transactions on Computers:181-194.

Kotla, R., Alvisi, L., Dahlin, M., Clement, A. and Wong, E. (2007). "Zyzzyva: speculative byzantine fault tolerance." Proceedings of twenty-first ACM SIGOPS Symposium on Operating Systems Principles:45-58.

Lamport, L., Shostak, R. and Pease, M. (1982). "The Byzantine Generals Problem." ACM Transactions on Programming Languages and Systems 4(3):382-401.

Luiz, A., Bessani, A., Lung, L. and Filgueiras, T. (2008). "RePEATS-Uma Arquitetura para Replicação Tolerante a Faltas Bizantinas baseada em Espaço de Tuplas." XXVI Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos, 2008.

Lung, L. C. (2009). Tolerância a Intrusões em Sistemas de Computação Distribuída. In II Seminário sobre Grandes Desafios da Computação no Brasil, pág. 13–16.

Reiser, H. and Kapitza, R. (2007). Hypervisor-Based Efficient Proactive Recovery. Proceeding of the 26th IEEE International Symposium on Reliable Distributed Systems (SRDS'07), Beijing, China.

Reiser, H. P. and Kapitza, R. (2008). Fault and Intrusion Tolerance on the Basis of Virtual Machines. Tagungsband des 1. Fachgespräch Virtualisierung. Germany.

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

Tsudik, G. (1992). "Message authentication with one-way hash functions." ACM SIGCOMM Computer Communication Review 22(5):29-38.

Veronese, G. S., et al. (2008). Minimal Byzantine Faul Tolerance. DI/FCUL Technical Report 08-30, December 2008.

Yin, J., Martin, J., 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.