Infraestrutura Baseada em Virtualização para Serviços Tolerantes a Intrusões

Jim Lau, Luciano Barreto, Joni da Silva Fraga
Departamento de Automação e Sistemas - DAS
Universidade Federal de Santa Catarina - UFSC
Caixa Postal 476 – 88040-900 – Florianópolis – Santa Catarina – Brasil
{jim, lucianobarreto, fraga}@das.ufsc.br

Resumo. *Neste artigo apresentamos uma infraestrutura que faz uso de virtualização cuja finalidade é fornecer suporte de tolerância a intrusões (faltas maliciosas ou bizantinas) para* serviços web. *No texto apresentamos uma abordagem que faz uso extensivo da tecnologia de virtualização e de memória compartilhada no sentido de conseguir custos mais baixos na execução de protocolos de máquina de estado (ME) em contexto bizantino. Nossa proposta permite que as requisições de clientes possam ser executadas com um número variável (entre f +1 e 2f+1) de réplicas. O artigo discute os algoritmos, apresenta detalhes de protótipo, testes e um confronto com trabalhos relacionados na literatura.*

Abstract. *This paper presents an infrastructure based in virtualization which provides support to intrusion tolerance (byzantine or malicious faults) for web services. The introduced approach makes extensive use of virtualization technology and shared memory in order to tolerate intrusions at a low cost of messages. The intrusion tolerance is typically achieved using state machine replication (SMR). Our approach allows client requests to be performed with a variable number (between f + 1 and 2f + 1) of execution replicas which is different from more classical implementations of SMR. This paper describes the algorithms, presents details of a prototype, testing and a comparative study with the related work in the literature.*

1. Introdução

A Internet tem sido um meio de comunicação de extrema importância para empresas estabelecerem e divulgarem seus serviços. Os *Service Providers* estão disseminados e fazem parte de nossas rotinas diárias. Esta profusão de serviços vem também acompanhada com preocupações. São preocupações normalmente ligadas à segurança. Embora toda a evolução experimentada pelas tecnologias de segurança, as invasões de provedores de serviço são fatos frequentes na Internet. A dificuldade em lidar com este problema é devido à complexidade sempre crescente dos suportes para aplicações disponíveis na rede mundial. Por mais cuidado que um projetista tenha no desenvolvimento destes sistemas, por mais que se acerque de técnicas de prevenção, vulnerabilidades sempre estarão presentes nestes sistemas e aplicações.

Na dificuldade de evitar acessos ilegais, muitas abordagens têm sido apresentadas na literatura no sentido de conter ou restringir as possíveis intrusões. Neste texto, temos interesse nos modelos de tolerância a intrusões para sistemas distribuídos [Correia 2005]. A ideia, neste caso, é manter o sistema distribuído evoluindo com comportamento correto mesmo diante de alguns componentes já corrompidos por intrusões. A ação maliciosa destes componentes invadidos não deve afetar o sistema

como um todo. Na literatura de *Dependability* [Laprie 1995], a noção de faltas bizantinas (ou faltas maliciosas) são usadas para descrever o comportamento de subcomponentes comprometidos de um sistema. E as soluções para estas corrupções parciais de um sistema envolvem o uso de técnica de replicação ativa, também conhecidas como replicação Máquina de Estado (ME) [Schneider 1990].

Em [Zhao 2007] e [Merideth et al. 2005] são introduzidos modelos de infraestrutura de serviços cuja preocupação é a tolerância a intrusões (faltas bizantinas) em provedores de serviços web. Estes modelos são construídos tomando como base replicações ME. O suporte para estas replicações nestes trabalhos fazem uso de algoritmos de acordo para ambientes bizantinos. Ambos trabalhos são baseados no PBFT (*Practical Byzantine Fault Tolerance* [Castro and Liskov 1999]). Nos modelos citados, cada réplica de serviço é executada em uma máquina física diferente. Estas replicações são implementadas em *clusters* ou redes locais devido aos custos dos protocolos de acordo e a dificuldade dos mesmos em lidar com as características das grandes redes (a complexidade em mensagens limita a habilidade dos mesmos em escalas maiores). O custo em mensagens destes protocolos, mesmo em execuções favoráveis (sem faltas maliciosas) assumem valores quadráticos, $O(N^2)$, sendo N o número de réplicas participantes nos protocolos de acordo e satisfazendo a condição $N \geq 3f + 1$ para garantir o acordo (execução correta do algoritmo de acordo), onde o valor de f corresponde ao limite de faltas ou intrusões admitidas no sistema distribuído de modo a não comprometer a correção do mesmo.

A motivação principal deste texto é apresentar uma alternativa aos trabalhos como os citados acima ([Zhao 2007] e [Merideth et al. 2005]). Neste sentido, introduzimos uma infraestrutura que faz uso extensivo de virtualização na implementação de ME em seu suporte de tolerância a intrusões (faltas maliciosas ou bizantinas) para serviços web. A abordagem de BFT (*Byzantine Fault Tolerance*) usada no nosso modelo de replicação fica restrita pela tecnologia de virtualização a acessos em memória compartilhada. O modelo, com o uso da virtualização, define um elemento confiável que vai determinar a separação de faltas de *crash* e faltas maliciosas, e a simplificação de nossos algoritmos. As réplicas de serviço construídas sobre máquinas virtuais ficam envolvidas exclusivamente com protocolos BFT (os mais custosos) limitados, portanto, a trocas via memória compartilhada. O nosso algoritmo de BFT, nestas condições, concretiza execuções de requisições de clientes com um número variável de réplicas (entre f +1 e 2f+1). Embora em termos de número de passos, o nosso algoritmo não tenha ganhos em relação aos algoritmos usuais, os custos se consideramos o *round-trip* requisição/resposta ficam extremamente reduzidos quando comparados com implementações em *clusters* ou redes locais. A infraestrutura de serviços implementada a partir de replicação ME faz uso também de conceitos de Orquestração Descentralizada e de padrões de *Web Services*, permitindo que provedores de serviço possam ser construídos de modo a tolerar intrusões sem ter que lançar mão de *clusters*, que é a solução usual encontrada na prática.

O presente artigo está dividido em 7 seções. A seção 2 apresenta as características e premissas assumidas no modelo de sistema usado, comportamento dinâmico deste modelo, descrito a partir do algoritmo BFT denominado IT-VM. Na sequência, a seção 3 descreve a infraestrutura de serviços que permite a construção de provedores de serviço tolerantes a intrusões. Os aspectos de implementação de um protótipo e a apresentação de resultados de testes e de comparações são apresentados na seção 4. Na

seção 5 são discutidos os principais trabalhos relacionados. Por fim, na seção 6 são apresentadas as conclusões finais do artigo seguido das referências na seção 7.

2. Modelo de Sistema

Nosso modelo é composto por dois grupos de processos que interagem através de comunicação pela rede: os servidores e os clientes. O conjunto de servidores, S= {S_0, S_1,..,S_{n-1}}, também chamados de réplicas de serviço, é formado por *n* processos com a mesma funcionalidade executados em máquinas virtuais. Os clientes, C= {C_0, C_1,..}, são processos que enviam requisições para os servidores através da rede.

No modelo, processos (clientes e servidores) são classificados como corretos ou faltosos. Processos corretos se comportam segundo as especificações do algoritmo, enquanto processos faltosos podem se comportar de maneira arbitrária desviando de suas especificações, podendo omitir mensagens, parar de responder, modificar o conteúdo das mensagens e fazer conluio com outros faltosos no sentido de dificultar a evolução correta da aplicação. Este tipo de comportamento é associado na literatura a faltas arbitrárias, bizantinas ou maliciosas (intrusões). Neste modelo de sistema assumimos um comportamento *parcialmente síncrono* [Dwork et al. 1988], ou seja, as interações entre processos clientes e o serviço no sistema distribuído admitem intervalos de assincronismo intercalados por períodos estáveis (síncronos). Estes períodos síncronos permitem a terminação destas comunicações em prazos de tempo definidos, porém não conhecidos. Os canais de comunicação são também assumidos no modelo como confiáveis. Canais de comunicação são ditos confiáveis quando mensagens não são criadas ou duplicadas e cada mensagem enviada por um processo P_i terminará por ser recebida pelo receptor P_j se o emissor (P_i) não for faltoso ou malicioso [Raynal 2005].

Usamos técnica de criptografia assimétrica para garantir a autenticidade das mensagens trocadas entre processos. A verificação das assinaturas envolve também o envio de certificados das chaves públicas dos assinantes nas mensagens trocadas.

2.1 Abordagem de Replicação do Modelo

As soluções para problemas de BFT (“*Byzantine Fault Tolerance*”) sempre tomam como base a técnica de replicação Máquina de Estado (ME) [Schneider 1990]. Em nossa proposta, na submissão de uma requisição de cliente, a implementação desta técnica inicialmente envolve um procedimento otimista com o uso de somente $f + 1$ réplicas respondendo a requisição. Porém, na detecção de um eventual desacordo entre as $f + 1$ respostas, novas réplicas são ativadas, levando o número de respostas de réplicas para $2f + 1$. O valor de f corresponde ao limite de faltas ou intrusões admitidas no sistema distribuído de modo a não comprometer a correção do mesmo. A nossa abordagem de BFT envolve também a identificação e substituição de réplicas corrompidas que provocaram a inconsistência (divergência de respostas) nas trocas com o cliente, retornando o número de réplicas para $f + 1$.

2.2 Virtualização

Conforme já citado, o objetivo desta proposta é dar suporte para serviços de modo que os mesmos sobrevivam a intrusões. No modelo de replicação, as réplicas invadidas e corrompidas não devem comprometer o atendimento das requisições pelo serviço

replicado. Para atender estes objetivos, a nossa abordagem de BFT é então construída através do uso da tecnologia de virtualização: os servidores são implementados e executados em separado, usando diferentes máquinas virtuais. Como nos fixamos no contexto de faltas maliciosas envolvendo réplicas de serviço comprometidas por intrusões, definimos o mapeamento das diversas máquinas virtuais que executam as réplicas em uma única máquina física servidora. Este enfoque implica em um modelo híbrido de faltas que separa os diferentes tipos falhas. Ou seja, em nível de máquinas virtuais tratamos exclusivamente com faltas bizantinas (maliciosas). Portanto, protocolos normalmente custosos serão restritos ao ambiente de uma máquina física. O problema de faltas de parada (*crash faults*) na máquina física servidora pode ser tratado com protocolos mais brandos em nível de sistema distribuído. Neste texto, nos limitamos à discussão de nossas soluções para faltas maliciosas.

O uso desta tecnologia de virtualização na implementação da nossa abordagem de BFT também oferece facilidades para o gerenciamento das diferentes réplicas do modelo. Como colocamos anteriormente, novas réplicas podem ser lançadas quando do desacordo das $f + 1$ respostas a uma requisição de cliente e, também, réplicas identificadas como corrompidas devem ser excluídas do modelo. Nesta tecnologia, máquinas virtuais podem ser facilmente iniciadas ou revogadas quando necessário.

Em nosso modelo, assumimos que o *hypervisor* e o VMM (*Virtual Machine Monitor*) podem apresentar vulnerabilidades, porém estas não podem ser exploradas externamente via rede ou pelas máquinas virtuais locais. Para garantir essa suposição, confiamos que o VMM forneça o isolamento adequado para a execução segura das máquinas virtuais. Esta é uma das premissas da tecnologia de virtualização [Chun et al. 2008] [Sahoo et al. 2010].

2.3 Memória Compartilhada

Uma memória compartilhada (*shared_mem*) é definida no modelo e assumida como um conjunto de registradores que armazenam valores e podem ser acessados por dois tipos de operações: escrita e leitura. Processos (réplicas e processos de suporte) utilizam estes registradores para se comunicarem. Essa memória compartilhada é oferecida e gerenciada pelo VMM. Políticas de acesso são definidas para que réplicas de serviço tenham acesso de leitura e escrita em áreas próprias desta memória, ou seja, cada máquina virtual só pode escrever em sua área particular para envio (nos *buffers* $reqBuffer_{Rj}$ e $repBuffer_{Rj}$). Já as leituras destas máquinas virtuais (réplicas) concorrem sobre dois *buffers* (*buffers* de leitura: *buffer_toExecute* e *buffer_toSend*) que tem como escritor um serviço de suporte: o *Agreement Service*. As políticas definidas para esta memória compartilhada são necessárias para garantir a integridade das requisições de clientes já ordenadas e das respostas correspondentes.

A correção nos acessos de memória compartilhada necessitam de garantias de *liveness*. Em [Fernandez et al. 2007] é introduzida à propriedade *AWB1*, que define que um processo não se comporta indefinidamente como uma execução assíncrona. Como consequência, a memória compartilhada por processos sempre permitirá acessos em tempo limitado, mas não necessariamente conhecido. Assumimos com isto que a memória compartilhada do modelo proposto, também possui um comportamento *parcialmente síncrono*.

2.4 Agreement Service

Um componente fundamental no nosso modelo é o *Agreement Service* (AS). Este serviço de suporte é responsável pela ordenação das requisições, verificação das respostas das réplicas de execução, geração e assinatura da resposta que será enviada para o cliente. Além disso, coordena a validação dos *checkpoints* feitos pelas réplicas de execução e a ativação de novas réplicas de serviço em caso de desacordo entre as réplicas nas respostas ou *checkpoints*. Ou seja, este componente trata de aspectos não funcionais, de forma que pode ser usado em qualquer tipo de aplicação sem modificações.

Este componente também pode ser considerado como uma abordagem de separação dos serviços de ordenação e de execução da aplicação, a exemplo do trabalho descrito em [Yin et al. 2003]. Desta forma, as réplicas de serviço que executam a aplicação, ficam expostas aos clientes via rede, enquanto o *Agreement Service* trata da ordenação total das requisições. O *Agreement Service* é mantido isolado e assumido como confiável no modelo.

O algoritmo IT-VM (*Intrusion Tolerance by Virtual Machines*), Algoritmo 1, descreve a dinâmica do modelo proposto. Inicialmente, é descrito o comportamento da execução sem réplicas discordantes (caso otimista). Seguindo, é apresentado o caso com falhas maliciosas (intrusões) e as discussões de aspectos de *checkpoint* e da recuperação da replicação.

2.5 Comportamento sem Desacordo

O comportamento otimista começa com um cliente C_i enviando uma requisição de serviço (operação *op*) às réplicas visíveis via rede em suas máquinas virtuais (linha1-3, do Algoritmo 1). É importante lembrar que a nossa abordagem de BFT envolve sempre inicialmente $f + 1$ réplicas ativas na execução de cada requisição proposta por clientes. Embora dispostas na mesma máquina física servidora, as VMs possuem diferentes endereços de rede. A requisição assinada é enviada, e juntamente encaminhado, o certificado de chave pública do cliente para posterior verificação ($Cert_{Ci}$, linha 2).

Na sequência, as réplicas de serviço, em suas VMs, recebem a requisição req_k e devem inserir uma cópia da mesma na memória compartilhada (*shared_mem*), em seus respectivos *buffers* $reqBuffer_{Rj}$ (linhas 8 e 9). O *Agreement Service* (AS) então será ativado por uma sinalização indicando a presença de mensagem nos *buffers* de requisição das réplicas (linha 26). Uma vez ocorrida a leitura de cópias de uma requisição req_k, o AS insere uma destas cópias em *shared_mem.buffer_toExecute* (requisições repetidas são descartadas). A posição ocupada neste último *buffer* da memória compartilhada vai determinar a ordem na sequência de execuções para esta requisição recém-inserida (linha 27 e 28). Esta inserção em *shared_mem.buffer_toExecute* deixa a mesma disponível para leituras das réplicas de serviço (leitoras deste *buffer*). O AS é o único escritor neste último *buffer*. Na linha 29 o AS salva o tempo atual para o controle de tempo ($timeout_k$) para a chegada de $f + 1$ respostas iguais para a requisição req_k.

Após definida a ordem de sequência de uma requisição, as réplicas são sinalizadas da presença da mesma no *buffer shared_mem.buffer_toExecute* (linha 10). A réplica R_j faz leitura da requisição ordenada (linha 11), e uma vez comprovada a sua autenticidade (verificação da assinatura linha 12), executa a operação *op* (linha 13). Na

sequência, R_j compõe com os resultados desta execução ($resp_k$) a mensagem de resposta rep_k (linhas de 14). Esta mensagem é inserida na memória compartilhada, em outro *buffer* de escrita da réplica ($shared\_mem.repBuffer_{Rj}$) na linha 15.

Com a escrita nos *buffers* $shared\_mem.repBuffer_{Rj}$, o AS é ativado por sinalização correspondente (linha 31). As respostas rep_k são então lidas e inseridas em *buffers* específicos para cada resposta de requisição $buffer(rep_k)$ (linhas 32 e 33). Ou seja, todas as respostas recebidas das réplicas para uma requisição req_k são inseridas em um mesmo *buffer* ($buffer(rep_k)$). Se o número de mensagens inseridas nestes *buffers* (específicos para respostas a uma dada requisição de cliente) atinge o limite *f + 1* e todas estas respostas são iguais (linhas 34 e 35), o AS cria então a sua assinatura sobre alguns campos da resposta e monta a resposta autenticada ($reply_k$) para ser enviada ao cliente C_i (linhas 36 e 37). Na sequência, o AS torna a mensagem $reply_k$ disponível no *buffer* *shared_mem.buffer_toSend* (linha 38). As condições verificadas nos testes das linhas 34 e 35 implicam em que não ouve desacordo sobre as respostas ao cliente C_i por parte de *f + 1* réplicas de serviço que estão ativas no processamento da requisição.

Algoritmo 1. Interações do Cliente C_i com o Serviço Replicado

```
{Cliente Ci}
1.  Sign_Ci ← Signature(<REQUEST, id_Ci, op >); //Assina a requisição
2.  req_k ← < REQUISIÇÃO, id_Ci, op, Sign_Ci, Cert_Ci>;
3.  send to all R_j ; //Envia para todas as réplicas
4.  upon Receive reply_k from R_j do
5.     if VerifySign(reply_k.Sign_AS, reply_k.Cert_AS ) then //Validação
6.           delivery(reply_k.resp); //Entrega para a aplicação
7.     end if

{Réplica Rj}
8.  upon Receive req_k at R_j
9.     shared_mem.reqBuffer_Rj .write (req_k); //Escreve na memória
10. upon ReqSignal at R_j do
11.    req_toExec ← shared_mem.buffer_toExecute.read();
12.    if VerifySign(req_toExec.Sign, req_toExec.Cert ) then
13.          resp_k ← thread.execute(req_toExec.op);
14.          rep_k ← < REPLY, id_Rj , id_Ci , resp_k >;
15.          shared_mem.repBuffer_Rj .write (rep_k );
16.    end if
17.  upon RepSignal at R_j do
18.       reply_j ← shared_mem.buffer_toSend.read();
19.       reply_j.id ← id_Ri //Adiciona o id do servidor na mensagem
20.       send reply_kj to C_i ; //Envia para o cliente
21.    upon (t() - δ_p ) ≥ period_Ck at R_j do
22.       state_kj ← thread_Checkpoint(); //Salva o estado
23.       hash_kj ← calculateHash(state_kj); //Gera o hash
24.       shared_mem.buffer_Checkpoint_Rj .write( hash_kj);
25.       δ_p ← t() ; //Inicia um novo período

{Agreement Service AS}
26. upon ReqSignal at AS do
27.    while ∀j, ∃(req_k) ∈ shared_mem.reqBuffer_Rj do //Lê as requisições da memória
28.       shared_mem.buffer_toExecute .write( req_k); //Escreve a requisição ordenada
29.       δ_k ← t() ; //Inicia um timeout para o recebimento das respostas
30.    end while
31. upon RepSignal at AS do
32.    while ∀j, ∃(rep_k) ∈ shared_mem.repBuffer_Rj do
33.       buffer(rep_k) ← buffer(rep_k) ∪ {shared_mem. repBuffer_Rj .read()};
34.       if |buffer(rep_k) | ≥ f +1 then //Se houverem f + 1 respostas
35.          if match (buffer ( rep_k)) = f +1 then //Se repostas forem iguais
36.             Sign_AS ← Signature(< REPLY, rep_k.id_Ci , rep_k.resp>);
37.             reply_k ←<REPLY, null, id_Ci , resp_k , Sign_AS , Cert_AS>;
38.             shared_mem.buffer_toSend.write( reply_k);
39.             restoreVMs(f+1) ; // Restaura replicação para f+1 réplicas
40.          else
41.             initiatedVMs(f, bufferHash ); //Inicia f novas réplicas
42.          end if
43.       end if
44.    end while
45. upon (t() - δ_k ) ≥ timeout_k at AS do //Timeout das respostas
46.    initiatedVMs(f, bufferHash ); //Inicia f novas réplicas
47.    δ_k ← t() ;
48. upon CkPointSignal at AS do //Armazena os hashs dos Checkpoints
49.    while ∀j, ∃(hash_kj) ∈ shared_mem.buffer_Checkpoint_Rj do
50.       bufferHash ← bufferHash ∪ {shared_mem.buffer_Checkpoint_Rj .read()};
51.    end while
```

A réplica R_j é ativada pelo sinal *RepSignal* (linha 17), lendo então $reply_k$ (resposta assinada para a requisição req_k) em *shared_mem.buffer_toSend*. Adiciona o seu identificador id_{Rj} nesta mensagem e envia a mesma para o cliente (linhas 18 a 20). Basta que uma destas mensagens $reply_k$ atinja o cliente e tenha a sua validade comprovada pela assinatura e certificado do AS que acompanham a mesma. Neste ponto o cliente pode fazer o *delivery* da resposta ($resp_k$) do processamento *op* solicitado para aplicação (linhas 4 a 7).

2.6 Comportamento com Desacordo em Respostas

No item anterior foi apresentado o caso otimista em que uma requisição é executada em *f + 1* réplicas e no qual não ocorre qualquer tipo de desacordo ou falha nas respostas destas réplicas. Mas quando da presença de réplicas maliciosas na replicação BFT as inconsistências ou desacordos vão ocorrer e devem ser detectados pelo *Agreement Service*. A não verificação das condições dos testes das linhas 34 e 35 ou o decurso do

prazo δ_k + *timeout* (linha 45) são os mecanismos de detecção do desacordo e do comportamento malicioso de réplicas do sistema.

Então, para a terminação do algoritmo, novas réplicas são necessárias. A não verificação do acordo e em tempo hábil provoca no Algoritmo 1, através do AS, a ativação de *f* novas réplicas (linhas 41 e 46). Ou seja, o número de VMs (réplicas) que devem processar req_k é então incrementado com *f* novas réplicas passando para o total de *2f +1* VMs executando a replicação do serviço. Com isto, mesmo que se tenham *f* máquinas virtuais corrompidas no desacordo inicial, a ativação destas novas réplicas fornece ao serviço redundância suficiente para fazer o mesmo evoluir o processamento e responder ao cliente. A ativação de novas VMs é concretizada nas linhas 41 ou 46 do Algoritmo 1 (chamada da função *initiatedVMs()*). Nesta chamada são passados o número de réplicas necessário (*f*), e uma lista de *hash (bufferHash)* que apresentaram acordo entre *f + 1* réplicas que se encontravam em execução. Essas novas réplicas buscam o estado da aplicação nas réplicas de execução e verificam se o mesmo é referente aos valores de *hash* enviados pelo *Agreement Service*. Se o estado for válido a nova réplica iniciada recupera o mesmo, executa as requisições necessárias para chegar à requisição que apresentou a falha para finalmente resolver o desacordo gerado.

As réplicas novas quando ativadas apresentam o mesmo comportamento indicado pelo Algoritmo 1. Começarão pela sinalização *ReqSignal* da linha 10, devido a presença de req_k em *shared_mem.buffer_toExecute* que ainda não foi processada por estas novas réplicas. O número de *2f +1* (e com o limite de *f* réplicas corrompidas) vai garantir então a obtenção do número de *f* +1 réplicas em "*match*" (com respostas iguais, linha 35) e, portanto a terminação do algoritmo.

O cliente neste modelo também tem papel ativo. Caso o cliente não receba uma resposta para sua requisição dentro de prazo definido ou as respostas recebidas são corrompidas, o mesmo retransmite a requisição. Os servidores que receberem a requisição assumem o comportamento normal do Algoritmo 1, nas linhas 8 e 9, e inserem a requisição retransmitida em seus respectivos *buffers* $reqBuffer_{Rj}$. O AS detecta então a retransmissão e ativa novas VMs incrementando a replicação para *2f + 1* réplicas. Uma vez ativadas, as novas réplicas vão se comportar como descrito paragrafo anterior.

Concluído o processamento e a terminação do algoritmo, o número de *2f + 1* VMs (réplicas) é diminuído para *f + 1* com também a identificação e a remoção das réplicas consideradas comprometidos (*restoreVMs* linha 39) e que levaram ao desacordo nas respostas na configuração inicial do servidor (*f + 1* réplicas iniciais).

2.7 *Checkpoint* das *Threads* de Execução

Periodicamente as réplicas executam a operação de *checkpoint* e criam um *snapshot* do estado da aplicação. Para isso é verificado decurso do prazo δp + *timeout* na linha 21 do Algoritmo 1. Então a operação de *checkpoint* é executada (linha 22) e um *hash* desse estado é gerado (linha 23). Esse valor de *hash* é enviado para a memória compartilhada através do *buffer shared_mem.buffer_Checkpoint* (linha 24) e inicia um novo período de decurso (linha 25). O AS aguarda *f + 1 hashes* iguais para *checkpoint* correspondente e então salva esses valores em um conjunto, que será utilizado na recuperação de uma réplica nova (linhas 48 a 51).

2.8 Algumas Considerações sobre o Comportamento Dinâmico do Modelo

Sobre a retransmissão do cliente, precisamos ainda esclarecer detalhes de modelo. Se no modelo de sistema distribuído tivéssemos assumido o comportamento de *fair links* [Raynal 2005], mensagens poderiam ser perdidas e as reposições de réplicas levariam a novas retransmissões de respostas. O cliente ficaria esperando uma resposta válida para evoluir. Em canais confiáveis que é a hipótese assumida no nosso modelo de BFT, as comunicações das réplicas sempre devem chegar e, dentro do prazo, se as condições favoráveis de sincronismo ocorrem dentro do sistema (premissa de sistema distribuído parcialmente síncrono). Uma resposta chegando dentro do prazo faz com que o cliente finalize a operação.

Em qualquer situação de desacordo em respostas ou *checkpoint* na configuração inicial (*f + 1* réplicas), novas réplicas são iniciadas. Porém, para poder responder ao cliente estas novas réplicas ao serem iniciadas precisam obter o estado anterior à requisição cujas respostas não atingiram o quórum exigido. Para isso, a réplica ao ser iniciada lê em memória compartilhada qual é o *checkpoint* estável ($state_{k-1}$) e seu *hash* correspondente, e se atualiza ficando pronta para iniciar sua operação normal.

O serviço replicado inicia sempre com *f + 1* réplicas, podendo sempre em situações de desacordo (em respostas ou *checkpoint*) aumentar o número de réplicas para *2f + 1*. As réplicas que contribuíram para o desacordo podem ser identificadas pelo AS nas comparações de respostas ou de *hashes*. As réplicas que estiverem em desacordo com a maioria são desativadas, visto que as mesmas poderão influenciar no desempenho do algoritmo e porque estão fazendo uso dos recursos dos servidores físicos de forma inadequada. No final do processamento da requisição req_k e do envio da resposta $reply_k$ ao cliente, mesmo que o limite *f* de maliciosas não tenha sido atingido na configuração de *2f + 1* acionada por desacordo, a replicação é reduzida sempre para a configuração inicial de *f + 1* réplicas. As réplicas que vão compor a nova configuração inicial estavam entre aquelas que produziram o valor mais frequente (de resposta ou de *checkpoint*).

Mesmo neste último caso, com canais confiáveis, réplicas poderiam se negar a enviar ao cliente suas respostas autenticadas pelo *Agreement Service* ($reply_k$). A redundância da configuração inicial (*f + 1*) já suficiente que pelo menos uma mensagem correta possa atingir o cliente em condições favoráveis de sincronismo. Os canais confiáveis garantem isto. As réplicas maliciosas que se negaram a transmitir $reply_k$ só poderão ser detectadas se ocorrer à retransmissão da requisição do cliente.

Visto que a memória compartilhada possui nos seus diversos *buffers* capacidade de armazenamento finito é necessário que se faça a remoção das informações antigas. A vinda de uma nova requisição de um cliente que supere uma anterior é um dos mecanismos usado pelo *AS* para ativar o coletor de lixo. Além disso, também foi especificado um *deadline* no sistema para impedir o problema de *buffer* infinito.

3. Serviços Replicados

A replicação Máquina de Estados (ME) que permite a tolerância a intrusões, descrita nas seções anteriores, é neste item no qual é apresentado na forma de serviços. Uma infraestrutura de serviços é definida para incorporar os conceitos e algoritmos do IT-VM. Esta infraestrutura identificada como *BFTWS-Orch* faz uso da composição de serviços web para implementar o nosso modelo de réplicas ativas. A composição de

réplicas é fundamentada no conceito de *Orquestração Descentralizada* [Chafle et al. 2004] que mantém fluxos de execução evoluindo de forma descentralizada na coordenação de composições. A literatura tem sido profícua na discussão desta coordenação de composições por fluxos descentralizados. Em [Chafle et al. 2004] são apresentados mecanismos para o particionamento do fluxo de execução de uma composição em vários *engines* distribuídos. A motivação destes e de outros autores em suas propostas é diminuir a concentração de todo o fluxo de controle de uma aplicação distribuída sob uma entidade única (um coordenador geral que é caracterizado por um único *engine*), encontrada em modelos convencionais de orquestração.

Em nosso modelo, portanto, *engines* distribuídos definem os fluxos de controle e de dados através dos serviços replicados na concretização das trocas de mensagens entre o cliente e as aplicações servidoras. É ilustrado na Figura 1 uma composição de serviço web através do framework *BFTWS-Orch*, onde as réplicas executam a mesma requisição de operação. Cada réplica de serviço está associada a um *engine* da orquestração distribuída, que executa a parte que lhe cabe do protocolo BFT de replicação ativa, que neste caso é o IT-VM. Ou seja, os fluxos de execução do protocolo citado foram estruturados através da linguagem BPEL [Arkin e Askary 2004], permitindo assim uma orquestração descentralizada da ME tolerante a intrusões. Os *engines* da Figura 1 que fazem parte do modelo definem uma estrutura hierárquica de composições que interagem entre si apresentando a ideia de uma orquestração distribuída. O *engine* cliente pode ser interpretado como uma composição de um nível (hierárquico) superior em relação às composições que concretizam as réplicas do serviço. Esta infraestrutura de serviços que concretiza o nosso modelo é realizada sobre um conjunto de máquinas virtuais e usa mecanismos de memória compartilhada para se comunicarem como enfatizado anteriormente neste texto.

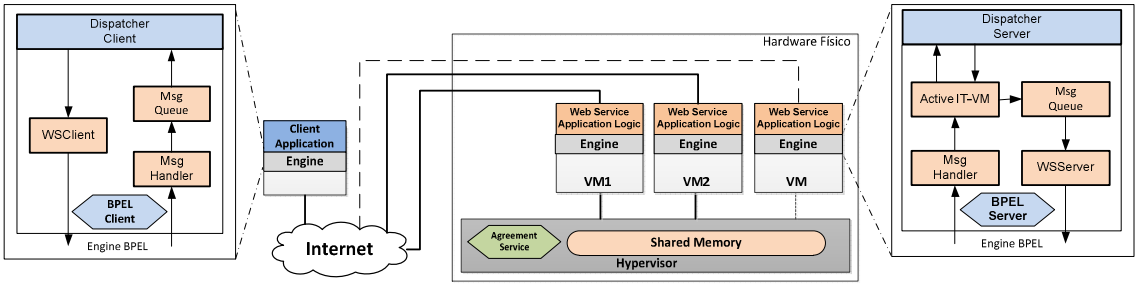


Figura 1 – Framework BFTWS-Orch

3.1 Descrição dos Componentes e suas Interações no BFTWS-Orch

Cada *engine* do *framework BFTWS-Orch* contém os mecanismos necessários para gerenciar as interações do mesmo com o *engine* do cliente e com a memória compartilhada (através do ambiente virtualizado). De forma mais precisa, podemos dizer que um *engine* é composta por uma unidade geradora de fluxo e de componentes que distribuem estes fluxos através de interações com o ambiente em que o *engine* atua. As interações de componentes de um *engine* devem então atuar com o ambiente de virtualização (*Agreement Service* e memória compartilhada) e coordenar a réplica correspondente na execução das requisições e envio de respostas ao cliente. É ilustrado na Figura 1 alguns destes componentes. No lado do cliente, temos o *BPEL Client*, que é o componente responsável pela geração de parte do fluxo de controle que envolve a execução de uma requisição de um cliente no ambiente distribuído. É através deste

BPEL Client que o *engine* no cliente interage com os componentes *Msg Handler*, *Msg Queue*, *WSClient*, *Dispatcher Client*. Todos estes últimos componentes quando ativados pelo *BPEL Client* devem atuar com o ambiente do cliente. Uma descrição mais específica destes componentes é apresentada na Tabela 1.

No lado servidor, temos os *engines* das réplicas que também são formadas por um componente principal identificado como *BPEL Server*. Este componente é o gerador de fluxo que ativas os blocos de interação do *engine* de réplica. A evolução deste fluxo define então as invocações do serviço de aplicação e as interações relacionadas com a execução do protocolo IT-VM. Os componentes de interação deste *engine* de réplica devem, portanto envolver interações implementando as comunicações via memória compartilhada, chamadas de operações na réplica de serviço e para a concretização do envio de respostas. Os componentes que fazem parte das *engines* de réplicas são: *Active IT-VM*, *Msg Queue*, *WSServer*, *Msg Handler*, *Dispatcher Server*. Os mesmos também estão descritos na Tabela 1.

Tabela 1: Componentes Internos de Engines

BPEL Client	É responsável pela execução do fluxo de controle para a execução remota de requisições do cliente. As sinalizações deste componente para outros componentes internos no cliente envolvem a montagem das requisições, a assinatura da mesma e a recepção de respostas e suas verificações.
Dispatcher Client	Componente que fornece uma interface entre a aplicação cliente e o *engine*. Tem a função de encapsular informações para realizar as invocações nas réplicas e obter do *Msg Queue* as respostas para serem entregues para a aplicação.
WSClient	Componente responsável em obter informações do serviço remoto (referência do serviço web, o método a ser executado e os parâmetros necessários para a sua invocação nas réplicas). Este componente é também responsável pela invocação transparente das múltiplas instâncias de Serviços Web com quem o cliente interage.
BPEL Server	É responsável pela execução do fluxo de controle através da sinalização a outros componentes do *engine* de réplica de serviço na comunicação com os clientes. Também gerência as trocas das réplicas, com os mecanismos de memória compartilhada definidas pelo protocolo IT-VM.
WSServer	Este componente é responsável em obter uma resposta assinada no *Msg Queue* e a prepara a resposta correspondente a ser enviada ao cliente.
Active IT-VM	Este componente atua na execução do protocolo IT-VM juntamente na comunicação com a área de memória compartilhada. Após receber uma notificação do *Msg Handler* executa as operações no qual envolve a ordenação (sinaliza um notificação para o *Dispatcher Server*), inserção da resposta (executada após receber uma resposta do *Dispatcher Server*) e validação e assinatura da resposta. A resposta assinada é transferida para o *Msg Queue* e uma notificação é enviada para o *WSServer* aguardando para ser processada.
Msg Queue	*Buffers* onde requisições são armazenadas antes de serem consumidas pelas réplicas correspondentes ou para armazenamento das respostas. Estes *buffers* são também usados nas eventuais retransmissões de mensagens.
Msg Handler	A função destes tratadores envolve a geração de mensagens de reconhecimento e pedidos de retransmissão. No cliente quando uma mensagem é recebida uma notificação é envida para *BPEL Client* e o componente *Dispatcher Client* e transferida para o *Msg Queue* onde aguarda pelo processamento. No lado servidor, o tratador sinaliza a chegada de requisição para o *BPEL Server* e a mesma é enviada para o *Active IT-VM*.

Os fluxos de execução definidos a partir dos componentes *BPEL Client* e *BPEL Server* têm, portanto, a função de interagir com os componentes internos dos *engines* cliente e de réplica, respectivamente. É através destes fluxos que os demais componentes internos de *engines* são chamados para exercerem suas funções. Na montagem de uma requisição, o *BPEL Client* do *engine* cliente interage com os componentes *Dispatcher Client* (fornece a interface entre o *engine* e o cliente) e *WSClient* para assinar a requisição e encapsular os parâmetros para execução da operação e o correspondente envio. Enquanto, no lado do servidor, o fluxo do *engine* (*BPEL Server*) da réplica irá interagir com os componentes para o recebimento da requisição que passa pelo componente *Msg Handler* e encaminha a requisição para o *Active IT-VM* (que executa as operações relacionadas ao protocolo IT-VM na parte de

memória compartilhada). Após a ordenação via *Active IT-VM* a requisição é encaminhada para o componente *Dispatcher Server* (fornece a interface entre o *engine* e a réplica) para ser executada pela aplicação. Ao final uma resposta é novamente encaminhada para o *Active IT-VM* que irá assinar a resposta e transferir para o *Msg Queue*. Uma sinalização será enviada para o *WSServer* e o *Dispatcher Server* informando que um resposta assinada se encontra para ser consumida. O *WSServer* busca a resposta assinada e envia para o cliente.

4. Protótipo e Testes

Um protótipo de serviço replicado foi implementado usando a tecnologia de *Web Service*s visando testar as nossas proposições. Neste protótipo cliente e servidor são executados sobre uma rede local de 100 Mbps composta por uma máquina servidora (máquina física) com processador *Core I7*, 8 GB de RAM.

A camada de virtualização do nosso modelo utiliza o *hypervisor* XEN [Barham 2003], a qual oferece um domínio especial, denominado *Domínio0,* que foi utilizado na nossa implementação como domínio confiável. As máquinas virtuais executam diferentes versões do Sistema Operacional Linux atendendo os requisitos de diversidade necessários para se conseguir a independência das vulnerabilidades das réplicas.

O framework *BFTWS-Orch* foi implementado fazendo uso do Apache ODE [Apache 2009]. Os softwares utilizados do *Apache ODE* permitiram a criação das orquestrações necessárias para execução do nosso modelo. A modelagem dos fluxos de execução foi descrita através da linguagem BPEL. Toda a estrutura de *engines* que o cliente necessita e também informações de acesso a este serviço podem ficar disponível a partir de um UDDI como usualmente se faz para qualquer composição de serviços. O restante das implementações foram feitas utilizando a linguagem Java.

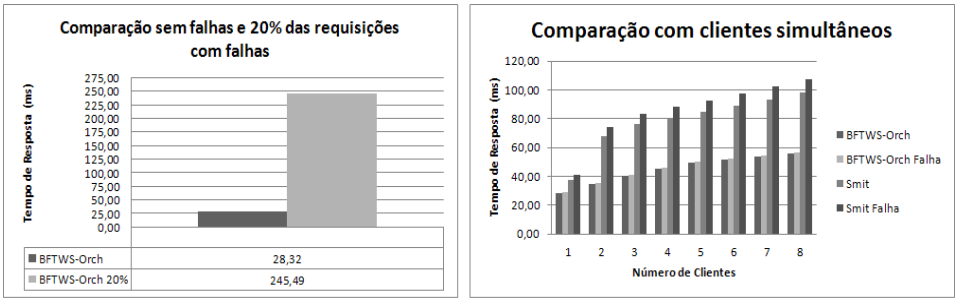
Uma aplicação bancária foi implementada neste protótipo onde operações de crédito e de débito são recebidas pelo serviço replicado em favor do cliente. As requisições destas operações e suas respostas são assinadas e validadas usando criptografia assimétrica RSA de 1024 bits.

Nos vários testes realizados sobre o protótipo desenvolvido, foi considerado o limite para réplicas maliciosas como $f = 1$. Este valor de f foi escolhido pela falta de recursos da máquina física hospedeira para manter muitas máquinas virtuais. Os testes realizados foram executados em dois cenários distintos: (i) ambiente sem réplicas maliciosas, caso otimista, e (ii), ambiente com réplicas maliciosas. No cenário com réplicas maliciosas 20% das requisições enviadas envolvem execução de réplicas maliciosas e, portanto, apresentam desacordo, ou seja, a cada 5 requisições o processo de recuperação de uma nova réplica foi executado.

A Figura 2.a demonstra o tempo médio de resposta no cliente nos dois cenários. Enquanto no cenário sem falhas o tempo de resposta foi *28,32 ms* no cenário com falhas esse tempo médio de resposta passou a ser de *245,49 ms.* Esse aumento se deve ao fato de que a cada requisição onde ocorre o desacordo das respostas, f novas réplicas devem ser ativadas para resolver essa falha. Além da ativação dessas f novas réplicas, existe ainda o processamento da identificação da réplica maliciosa e a remoção da mesma. Todo esse processo de recuperação do protocolo tem tempo médio de *1100 ms.* Ainda que o tempo médio de resposta no cenário com falhas tenha um valor extremamente

maior que no cenário otimista, o sistema continuou respondendo corretamente para o cliente mesmo com uma grande presença de falhas.

A Figura 2.b ilustra uma comparação do tempo de resposta considerando a utilização por usuários simultâneos. Neste cenário comparamos a nosso modelo de BFT com a abordagem SMIT que é muito similar ao nosso por usar máquinas virtuais. A abordagem SMIT é descrita na seção de trabalhos relacionados. Neste cenário da Figura 2.b, comparamos os tempos médios de resposta dos casos com falhas e sem falhas de ambas abordagens. O tempo de resposta é impactado em todos os testes deste cenário, visto que com o aumento do número de clientes, a utilização e concorrência sobre os recursos da máquina física são incrementadas. A nossa abordagem tem um melhor desempenho em termos de tempo de resposta. Isso se deve ao fato de envolver configurações com menos réplicas (*f+1*) e, portanto, menos VMs. Com o número de réplicas reduzido, a possibilidade de uma melhor utilização dos recursos do servidor físico melhora o desempenho da replicação. Outro ponto em que a nossa abordagem apresenta vantagens é na ocorrência de falhas maliciosas. Enquanto o SMIT, na ocorrência de falhas, as réplicas comprometidas continuam na execução do protocolo, influenciando no desempenho do mesmo, em nosso modelo as réplicas maliciosas são continuamente removidas, de forma que o protocolo sempre seja reconfigurado para ser executado com somente *f + 1* réplicas corretas.



(a) (b)

Figura 2 – (a) Comparação da arquitetura com falhas e sem falhas (b) Tempo de resposta considerando usuários simultâneos.

5. Trabalhos Relacionados

Na literatura são encontrados alguns trabalhos utilizando o conceito de virtualização como suporte para algoritmos distribuídos tolerantes a faltas bizantina. Entre estes trabalhos temos [Reiser and Kapitza 2007], [Júnior et al. 2010] e [Wood et al. 2011]. Em [Reiser and Kapitza 2007] é proposto a execução de um serviço replicado implementado sobre várias máquinas virtuais também em um único sistema físico. Além disso, esta proposta faz uso de um componente confiável para auxiliar na execução do algoritmo. Entretanto, este componente se encontra exposto na rede possibilitando uma invasão, o que pode comprometer o correto funcionamento do modelo.

Em [Júnior et al. 2010] é proposta a arquitetura SMIT e um algoritmo onde as réplicas de serviço são executadas em máquinas virtuais e a comunicação entre as réplicas é realizada através de uma abstração de uma memória compartilhada. Este modelo, pelo uso de memória compartilhada e elemento confiável tem também a redução no número necessário de réplicas de *3f + 1* para *2f + 1* quando em comparação com PBFT executados sobre redes locais [Castro and Liskov 1999]. No SMIT, as

intrusões vão sendo acumuladas no ciclo de vida do sistema. Não existe recuperação da replicação diante de réplicas maliciosas.

Em [Wood et al. 2011] é proposta a arquitetura ZZ e um protocolo utilizando a tecnologia de virtualização para execução de um protocolo BFT no qual aplica o conceito de separação do protocolo de acordo e da execução definida por [Yin et al. 2003]. Neste trabalho existe também a alteração dinâmica da replicação do modelo, com a replicação também assumindo configurações de $f+1$ e $2f+1$ réplicas. As diferenças do ZZ com a nossa abordagem de BFT é que as VMs de réplicas no ZZ são executadas em um conjunto de máquinas físicas. O ZZ não trabalha com a ideia de elementos confiáveis como o AS do nosso modelo e, portanto, não permite a separação de faltas *crash* e de intrusões. No ZZ, as VMs não interagem via memória compartilhada, mas sim usando troca de mensagens via protocolos de rede o que implica em custo adicional de interações entre VMs.

Em relação aos trabalhos envolvendo serviços web e replicação Máquina de Estados para a tolerância a intrusões, já citamos neste texto os trabalhos de [Zhao 2007] e [Merideth et al. 2005]. Estes trabalhos introduzem infraestrutura visando à tolerância a intrusões em provedores de serviços web. Estes trabalhos são baseados no PBFT [Castro and Liskov 1999] onde utilizam $3f + 1$ réplicas para garantir a execução do algoritmo de acordo. Neste trabalho, cada réplica de serviço é executada em uma máquina física no qual eleva o custo de mensagens da replicação para $O(N^2)$, sendo N o número de réplicas no modelo. No nosso modelo o custo de mensagens na rede cai para $O(f+1)$.

6. Considerações finais

Neste artigo apresentamos uma infraestrutura de serviços que faz uso de virtualização na sua finalidade de fornecer suporte de tolerância a intrusões (faltas maliciosas ou bizantinas) para provedores de serviço. No texto, apresentamos a nossa abordagem de BFT que faz uso extensivo da tecnologia de virtualização e de memória compartilhada no sentido de conseguir custos mais baixos na execução de protocolos de máquina de estado (ME) em contexto bizantino. Na verdade, esta abordagem (IT-VM), por enfatizar a separação de faltas de *crash* e faltas maliciosas, limita os protocolos de ME que tratam da tolerância a intrusões. Esta separação e também o uso da virtualização, permitem que as requisições de clientes possam ser executadas com um número variável (entre $f+1$ e $2f+1$) de réplicas. Esta infraestrutura de serviços é implementada com o uso de conceitos de Orquestração Descentralizada e padrões de *Web Services*, permitindo que Provedores de Serviço possam ser construídos de modo a tolerar intrusões sem ter que lançar mão de *clusters* que é a solução usual encontrada na prática.

O algoritmo de BFT que definimos neste sistema tem uma dependência muito forte em relação ao elemento confiável introduzido no nosso modelo: o *Agreement Service* (AS). Se este elemento for violado o sistema como um todo deixa de funcionar corretamente. A correção deste elemento pode ser verificada facilmente devido à simplicidade de suas funções. O isolamento deste componente confiável é garantido na nossa abordagem porque o mesmo fica num domínio único, isolado das máquinas virtuais que executam as réplicas e também inacessível através da rede. Na implementação com o XEN usamos o *domínio 0* para o seu isolamento.

7. Referências

Apache (2009). Apache ODE. *Disponível em: http://ode.apache.org/.*

Arkin, A. and Askary, S. (2004). WS-BPEL: Web Services Business Process Execution Language Version 2.0. OASIS Open, December.

Barham, P., Dragovic, B., Fraser, Keir, et al. (2003). Xen and the Art of Virtualization. *In* (SOSP'03), v. 37, n. 5, p. 164.

Castro, M. and Liskov, B. (1999). Practical Byzantine Fault Tolerance. *In* (USENIX'99), v. 20, n. 4, p. 398-461.

Chafle, G. B., Chandra, S., Mann, V. and Nanda, M. G. (2004). Decentralized Orchestration of Composite Web Services. *In* (WWW'04), p. 134-143.

Chun, B., Maniatis, P. and Shenker, S. (2008). Diverse replication for single-machine byzantine-fault tolerance. *In* (*USENIX'08)*, p. 287-292.

Correia, M. P. (2005). Serviços Distribuídos Tolerantes a Intrusões: Resultados Recentes e Problemas Abertos. *Technical Report* - TR'05, p. 113–162.

Dwork, C., Lynch, N. and Stockmeyer, L. (1988). Consensus in the Presence of Partial Synchrony. *In (JACM'88)*, v. 35, n. 2, p. 288-323.

Fernandez, A., Jimenez, E. and Raynal, M. (2007). Electing an Eventual Leader in an Asynchronous Shared Memory System. *In (DSN'07)*, p. 399-408.

Júnior, V. S., Lung, L. C., Correia, M., Fraga, J. da S. and Lau, J. (2010). Intrusion Tolerant Services Through Virtualization: a Shared Memory Approach. *In (AINA´10)*, p. 768-774.

Laprie, J. C. (1995). Dependable Computing and Fault Tolerance:Concepts and Terminology. *In IEEE Proceedings of FTCS*.

Merideth, M. G., Iyengar, A., Mikalsen, T., et al. (2005). Thema: Byzantine-Fault-Tolerant Middleware for Web-Service Applications. *In (SRDS'05)*, p. 131-142.

Reiser, H. P. and Kapitza, R. (2007). VM-FIT: Supporting Intrusion Tolerance with Virtualisation Technology. *In (WRAITS´07)*, p. 18-22.

Raynal, M. (2005). A Short Introduction to Failure Detectors for Asynchronous Distributed Systems. *In* (*SIGACT'05)*, v. 36, n. 1, p. 53-70.

Sahoo, J., Mohapatra, S. and Lath, R. (2010). Virtualization: A Survey on Concepts, Taxonomy and Associated Security Issues. *In (ICCNT'10)*, p. 222-226.

Schneider, F. B. (1990). Implementing Fault-Tolerant Approach: A Tutorial Services Using the State Machine. *Computing*, v. 22, n. 4.

Wood, T., Singh, R., Venkataramani, A., Shenoy, P. and Cecchet, E. (2011). ZZ and the Art of Practical BFT Execution. *In (EuroSys '11)* . ACM, p 123–138.

Yin, J., Martin, J.-P., Venkataramani, A., Alvisi, L. and Dahlin, M. (2003). Separating Agreement from Execution for Byzantine Fault Tolerant Services. *In (SOSP'03)*, v. 37, n. 5, p. 253-267.

Zhao, W. (2007). BFT-WS: A Byzantine Fault Tolerance Framework for Web Services. *Eleventh International IEEE Conference Workshop*, p. 89-96.