Tolerância a Faltas Bizantinas usando Registradores Compartilhados Distribuídos

Marcelo Ribeiro Xavier Silva[1], Lau Cheuk Lung[1], Aldelir Fernando Luiz[2,3], Leandro Quibem Magnabosco[1]

[1]Departamento de Informática e Estatística - Universidade Federal de Santa Catarina - Brasil

[2]Câmpus Avançado de Blumenau - Instituto Federal Catarinense - Brasil

[3]Departamento de Automação e Sistemas - Universidade Federal de Santa Catarina - Brasil

marcelo.r.x.s@posgrad.ufsc.br, lau.lung@inf.ufsc.br, aldelir@das.ufsc.br

leandro.magnabosco@posgrad.ufsc.br

Abstract. *State Machine Replication (SMR) is a technique commonly used in the implementation of distributed services that tolerates Byzantine Faults. Originally the approaches based on this technique did require $3f+1$ servers to tolerate f faults. Recently, through the use of a tamper-proof component, some approaches reduced this number to $2f+1$. In general, the construction of this components enforce some complex hardware and software modification in the server machines. We present Registered Paxos (RegPaxos), an approach that explores virtualization and shared memory emulation to simplify the creation of a tamper-proof component. With this architecture we were also able to reduce the latency, in means of the number of communication steps, that is only comparable to speculative algorithms.*

Resumo. *Replicação de Máquina de Estados é uma técnica comumente utilizada na implementação de serviços distribuídos que toleram faltas bizantinas. Originalmente as abordagens baseadas nesta técnica necessitavam $3f+1$ servidores para tolerar f faltas. Recentemente, através do uso de componentes confiáveis, algumas abordagens conseguiram reduzir este número para $2f+1$. Para construir estes componentes confiáveis é necessário fazer algumas modificações complexas nos servidores, tanto do ponto de vista de software quanto de hardware. Nós apresentamos o Paxos Registrado (RegPaxos), uma abordagem que explora tecnologias de virtualização e compartilhamento distribuído de dados para simplificar a criação da componente inviolável de tolerância a faltas. Com esta arquitetura fomos capazes também de alcançar uma latência (em números de passos para comunicação) comparável apenas com algoritmos especulativos.*

1. Introdução

A aplicação de conceitos de dependabilidade para construir sistemas distribuídos seguros tem crescido consideravelmente sob a designação de tolerância a intrusão ou tolerância a faltas bizantinas [Veríssimo et al. 2003, Correia et al. 2004]. Esse tema de pesquisa tem sido muito explorada nas últimas décadas e, dentre as abordagens utilizadas, destacam-se as que fazem uso de técnicas de redundância. Estas técnicas são implementadas através da replicação de máquina de estados (do inglês, *State Machine Replication* - SMR) [Lamport 1978, Schneider 1982] que combina uma série de mecanismos que contribuem

para a manutenção da disponibilidade e integridade dos serviços e aplicações, bem como dos ambientes de execução.

A abordagem SMR tem sido utilizada para tolerar faltas Bizantinas (do inglês, *Byzantine Fault Tolerant* - BFT) [Reiter 1995, Castro and Liskov 2002], afim de manter o funcionamento correto do sistema a despeito da ocorrência de faltas. Os algoritmos tolerantes a faltas bizantinas [Veríssimo et al. 2003] têm como objetivo permitir que os sistemas continuem operando dentro de suas especificações de funcionamento, mesmo que alguns de seus componentes apresentem comportamentos arbitrários [Castro and Liskov 2002, Reiter 1995, Yin et al. 2003]. Alguns trabalhos comprovam que através do uso destes algoritmos é possível projetar serviços confiáveis como sistemas de arquivos em rede, *backup* cooperativo, serviços de coordenação de autoridades certificadoras, banco de dados, sistemas de gerenciamento de chaves, etc [Veronese et al. 2011, Bessani et al. 2007].

Os algoritmos BFT, em sua maioria, necessitam de um mínimo de $3f + 1$ réplicas [Castro and Liskov 2002, Reiter 1995, Kotla et al. 2008] para tolerar f faltosas. Na realidade, quando usada para tolerar faltas de *crash*, a redundância de máquinas de estado necessita apenas $2f + 1$ réplicas [Schneider 1990]. O número adicional de réplicas é necessário para tolerar comportamentos maliciosos e/ou arbitrários [Correia et al. 2012]. O custo é ainda mais alto se considerarmos que a heterogeneidade é uma premissa importante neste tipo de sistema [Obelheiro et al. 2005]. Trabalhos recentes conseguem melhorar a resiliência dos sistemas BFT tolerando faltas com apenas $2f + 1$ réplicas. Estes trabalhos baseiam-se em componentes invioláveis criadas através de modificações em *hardware* [Veronese et al. 2011, Chun et al. 2007], *software* da máquina servidora [Correia et al. 2004, Veronese et al. 2011] e/ou rede de comunicação [Correia et al. 2004]. Existem também trabalhos que propõe o uso de tecnologias de virtualização para implementar seus sistemas tolerantes [Reiser and Kapitza 2007, Stumm et al. 2010]. Esta abordagem com certeza diminui o custo de replicação, mas transforma a máquina física num ponto único de falha.

Além da resiliência, outro fator importante para avaliação do desempenho de sistemas BFT é o atraso no processamento de uma requisição, ou latência [Correia et al. 2012]. A quantidade de passos de comunicação necessários durante uma execução na ausência faltas é considerada como métrica de latência nestes sistemas [Veronese et al. 2011]. Os algoritmos especulativos, em que os clientes participam ativamente do progresso do sistema, são os que possuem a menor quantidade de passos [Kotla et al. 2008, Veronese et al. 2011].

Neste artigo, apresentamos o RegPaxos que também necessita de apenas $2f + 1$ réplicas para tolerar f faltosas. A arquitetura de sistema proposta neste trabalho é baseado em um modelo híbrido em que variam, de componente para componente, as suposições de sincronismo e presença/severidade de faltas e falhas [Correia et al. 2002, Veríssimo 2006, Correia et al. 2004]. Neste modelo, a rede é separada em duas. A primeira, chamada de rede de *payload*, é assíncrona e utilizada para que os clientes se comuniquem com os servidores. A segunda é uma rede inviolável e é utilizada pelos servidores para ordenar as requisições dos clientes. Nesta revisita ao modelo híbrido proposto em [Correia et al. 2002, Veríssimo 2006], contribuímos com a redução da complexidade na construção da rede inviolável através do uso de dois artifícios, virtualização, para isolar o serviço confiável do serviço disponível aos clientes, e a utilização de um componente, aqui denominado Registradores Compartilhados Distribuídos (DSR - acrônimo para *Distributed Shared Register*), para ordenar requisições dos clientes. Além disso, o RegPaxos é o pri-

meiro algoritmo não especulativo capaz de equiparar-se com algoritmos especulativos em relação à latência.

2. Trabalhos Relacionados

O uso de máquinas de estados foi introduzido por Lamport em sistemas onde não se considerava a ocorrência de faltas [Lamport 1978]. Schneider generalizou esta abordagem considerando sistemas sujeitos a faltas de *crash*, o que serviu como base para o Rampart [Reiter 1995], e para outros sistemas tolerantes a faltas bizantinas.

A considerar o uso de virtualização como suporte para sistemas BFT, o VM-FIT [Reiser and Kapitza 2007] foi uma das primeiras propostas a aplicar virtualização para tolerar faltas bizantinas. Os autores propõe o uso de várias máquinas virtuais sobre uma máquina física para atender requisições de clientes. A ordem adotada para execução das operações de clientes vem de um *virtual machine manager* (VMM) adaptado para atuar como ordenador. Esta abordagem é importante por ser uma das pioneiras e por reduzir o custo de replicação. Entretanto, esta redução transforma a máquina física em um ponto único de falhas, além disso, a necessidade de se alterar o VMM dificulta seu desenvolvimento.

O facilitador para sistemas distribuídos, *Attested Append-only Memory* (A2M) apresentado em [Chun et al. 2007] provê uma abstração de um sistema de *log* confiável, através da qual é possível criar protocolos imunes a faltas. Através do uso do A2M é possível criar variações do sistema BFT apresentado por Castro e Liskov [Castro and Liskov 1998]. Com o A2M o sistema mantém suas propriedades de segurança e progresso, mesmo que metade das réplicas estejam faltosas. É uma abordagem simples em termos de entendimento e atinge a melhor resiliência prática. Entretanto, sua implementação requer o gerenciamento de cinco arquivos de *log*, aumentando a necessidade de armazenamento e o torna mais complexo do que seus autores assumiram.

Os algoritmos assíncronos e tolerantes a faltas bizantinas por replicação de máquina estados MinBFT e MinZyzzyva apresentados em [Veronese et al. 2011] melhoram algoritmos anteriores requerendo apenas $2f + 1$ ao invés $3f + 1$ réplicas. Os autores apresentam um serviço confiável simples. O algoritmo é muito simples e usa o *chip* TPM (*Trusted Platform Module*) como componente inviolável para implementar o serviço USIG (*Unique Sequential Identifier Generator*) que designa números de ordenação para as mensagens vindas dos clientes, assim, toda réplica correta executa as requisições na ordem designada pelo USIG. A contribuição deste trabalho vem em termos de custo, resiliência e complexidade de gerenciamento, ficando mais próximo dos algoritmos de replicação tolerantes a faltas catastróficas. Entretanto, para a criação do serviço inviolável são necessárias modificações na máquina servidora que precisa possuir o *chip* TPM, em nível de *hardware*, tornando sua implementação mais complexa do que utilizar-se de virtualização e de compartilhamento distribuído de dados.

Em [Correia et al. 2004] é apresentado o *Trusted Timely Computing Base* (TTCB) com Trusted Multicast Ordering (TMO). Neste trabalho é discutido um sistema de replicação tolerante a intrusão baseado no conceito de replicação de máquina de estados. Com o uso de um *wormhole*, um componente inviolável distribuído que pode ser implementado localmente na máquina servidora ou diretamente em *hardware*, os autores conseguiram alcançar a resiliência $\lfloor \frac{n-1}{2} \rfloor$. O sistema possui duas redes para tolerar faltas, a primeira é assíncrona e se estabelece entre os clientes e os servidores (rede de *payload*), enquanto

que a segunda é síncrona e é utilizada para que os servidores conectem-se ao *wormhole* [Correia et al. 2002] TTCB (*Trusted Timely Computing Base*. O SMR, proposto no artigo, usa o serviço TMO (*Trusted Multicast Ordering*) implementado dentro do TTCB. Por estar dentro do TTCB, o TMO tem sua execução assegurada contra faltas maliciosas. Seu funcionamento é simples, o serviço designa números de ordenação para cada mensagem recebida dos clientes, então, todas as réplicas corretas utilizam esta ordem para executar as requisições. A abordagem usando TTCB e TMO é de suma importância na área de replicação de máquina de estados já que consegue atingir a melhor resiliência prática.

Por outro lado, a implementação do TTCB requer o uso de um *hardware* à parte, como pode ser visto em [Correia et al. 2002]. Este componente dificulta a aplicação prática, além disso, é necessário proteger este componente de *hardware* com algum componente em *software*, responsável por separar as operações do TTCB das operações do sistema operacional. Os autores fazem essa proteção através da modificação do *kernel* do RT-Linux. Esta abordagem tem as seguintes implicações:

1. Dependência da versão do *kernel* que pode estar desatualizada ou mais vulnerável, se comparado às versões mais recentes. Com o RegPaxos isto não é um problema, pois não há restrição quanto ao uso de sistema operacional, permitindo heterogenenidade [Obelheiro et al. 2005].
2. As bibliotecas de acesso ao TTCB precisam ser implementadas para cada linguagem de programação. Isto limita a diversidade, já que diferentes paradigmas precisam ser adaptados para serem utilizados com o TTCB. O RegPaxos provê acesso independente de linguagem, permitindo que o serviço seja implementado através de qualquer paradigma ou linguagem de programação.
3. O TTCB impõe limitações a um atacante através da remoção do acesso de super usuário no sistema operacional, protegendo apenas o ID/GID 0 (*root*). Técnicas do tipo *privilege escalation* podem ser empregadas para explorar falhas de *design* como *buffer, stack* ou *heap overflow* e erros de configuração. No RegPaxos, o isolamento é feito através do gerenciador de máquinas virtuais (*hypervisor*) da máquina virtual, portanto, um atacante consegue penetrar apenas na máquinas virtuais, deixando o sistema hospedeiro protegido.

Além disso, no TTCB, a separação entre as duas redes é feita pelas interfaces de rede, o que implica na necessidade de pelo menos duas placas de rede por servidor. No RegPaxos a separação é feita através do modo *bridge* da tecnologia de virtualização, diminuindo a quantidade de recursos de *hardware* necessários.

Tabela 1. Comparação entre os protocolos em ambientes livres de falhas

Protocolos Avaliados	***Propriedades e características***			
	# Réplicas	**# Máquinas físicas**	**# Passos de comunicação**	**Especulativo**
PBFT [Castro and Liskov 1998]	$3f+1$	$3f+1$	5	não
Zyzzyva [Kotla et al. 2008]	$3f+1$	$3f+1$	3	sim
TTCB [Correia et al. 2004]	$2f+1$	$2f+1$	5	não
A2M-PBFT-EA [Chun et al. 2007]	$2f+1$	$2f+1$	5	não
MinBFT [Veronese et al. 2011]	$2f+1$	$2f+1$	4	não
MinZyzzyva [Veronese et al. 2011]	$2f+1$	$2f+1$	3	sim
RegPaxos	$2f+1$	$2f+1$	3	não

3. Visão Geral do RegPaxos

3.1. Modelo de Sistema

O modelo de sistema adotado é híbrido [Veríssimo 2006], o que significa que existe variação, de componente para componente, em relação às suposições de sincronismo e presença/severidade de faltas e falhas [Correia et al. 2002, Veríssimo 2006]. Em nosso modelo, consideramos diferentes suposições para os subsistemas que executam no *host* e no *guest* das máquinas virtuais que compõe o sistema. Neste modelo, o conjunto $C = \{c_1, c_2, c_3, ...\}$ representa o número finito de processos clientes e $S = \{s_1, s_2, s_3, ..., s_n\}$ representa o conjunto de servidores contendo n elementos que implementam o serviço oferecido pelo sistema SMR. Cada servidor possui uma máquina virtual que contém apenas um sistema como *guest*. O modelo de falhas admite que um número finito de clientes e até $f \leq [\frac{n-1}{2}]$ servidores incorram em faltas de suas especificações, apresentando comportamento arbitrário ou bizantino [Lamport et al. 1982]: um processo faltoso pode desviar de suas especificações omitindo ou parando de enviar mensagens, ou ainda apresentar qualquer tipo de comportamento (malicioso ou não) não especificado. Entretanto, assumimos independência entre as faltas, em outras palavras, a probabilidade de um servidor incorrer em faltas é independente da ocorrência de faltas em outro servidor. Na prática, é possível atingir-se isto através do uso extensivo de diversidade (i.e. diferentes ambientes de *hardware/software*, sistemas operacionais, máquinas virtuais, bases de dados, linguagens de programação, etc) [Rodrigues et al. 2001].

Nosso modelo de sistema prevê o uso de duas redes de comunicação. A primeira, a rede de *payload* é assíncrona e é utilizada para transferência de dados da aplicação. Não existem suposições baseadas em tempo para a rede de *payload* e sua utilização se dá entre clientes e servidores para envio de requisições e respostas. A segunda rede é controlada (registradores compartilhados distribuídos) e é utilizada para que os servidores troquem mensagens do protocolo de acordo. Assumem-se as seguintes propriedades para este rede:

- é segura, isto é, resistente a qualquer possível ataque, falhando apenas por *crash*;
- é capaz de executar operações com delimitação temporal;
- provê apenas duas operações, leitura e escrita em registradores. Estas operações não podem ser afetadas por faltas maliciosas.

Assumimos que cada par cliente-servidor c_i, s_j e cada par de servidores s_i, s_j está conectado por um canal confiável com duas propriedades: se o remetente e o destinatário de uma mensagem são ambos corretos, então (1) a mensagem é eventualmente recebida e (2) a mensagem não é modificada no canal [Correia et al. 2004]. Na prática, estas propriedades podem ser obtidas com o uso de criptografia e retransmissão [Wangham et al. 2001]. *Message authentication codes* (MACs) são *checksums* criptográficos que servem ao nosso propósito, e usam apenas criptografia simétrica [Menezes et al. 1996, Castro and Liskov 2002]. Os processos precisam compartilhar chaves simétricas para fazerem uso de MACs. Neste artigo, assumimos que estas chaves são distribuídas antes do protocolo ser executado. Na prática, isto pode ser resolvido usando protocolos de distribuição de chaves disponíveis na literatura [Menezes et al. 1996].

3.2. Arquitetura do Ambiente

A arquitetura do sistema está representada na figura 1. As máquinas virtuais são os servidores que possuem a réplica do serviço. As máquinas físicas compõe uma abstração de

memória compartilhada emulada [Guerraoui and Rodrigues 2006] (veja 3.4.) aqui chamada de registradores compartilhados distribuídos (DSR - acrônimo para *Distributed Shared Register*). Cada máquina física possui seu próprio espaço dentro dos DSR, onde sua respectiva máquina virtual registra mensagens do tipo PROPOSE, ACCEPT ou CHANGE. Todos os servidores podem ler todos os registros, não importando quem o registrou.

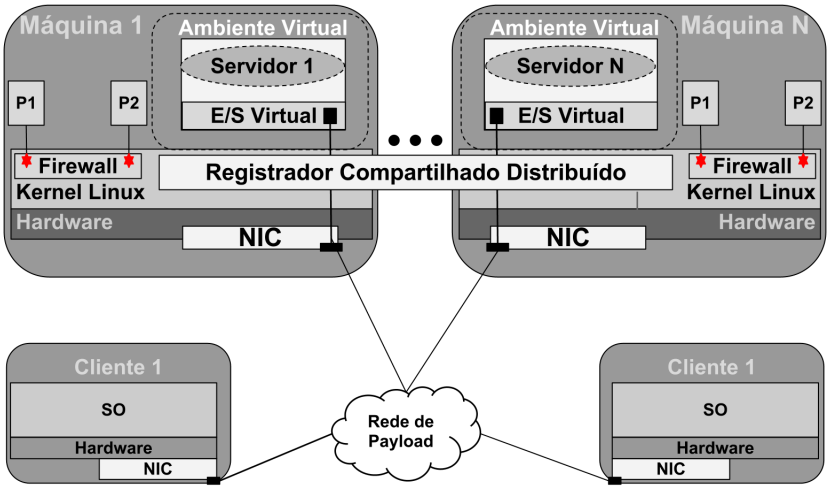


Figura 1. Visão geral da arquitetura

Nós assumimos que apenas as máquina físicas podem, de fato, conectar-se à rede utilizada pelos DSR. Isto implica que os registradores compartilhados distribuídos só estão acessíveis pelas máquinas físicas que hospedam (*host*) as máquinas virtuais. Transitivamente, os DSR não estão acessíveis através do *guest* das máquinas virtuais. Cada processo é encapsulado dentro de sua própria máquina virtual, assegurando isolamento. Todas as comunicações entre clientes e servidores acontecem em uma rede separada e, do ponto de vista do clientes, a máquina virtual é transparente. Isto significa que os clientes não reconhecem a arquitetura física-virtual. Cada máquina possui apenas uma interface de rede, um *firewall* e/ou o modo *bridge* são utilizados no *host* para assegurar a divisão entre as redes. Assumimos que as vulnerabilidades do *host* não podem ser exploradas através da máquina virtual. O gerenciador de máquinas virtuais (*hypervisor*) assegura este isolamento, garantindo que um atacante não tem meios para acessar o *host* através da máquina virtual. Isto é uma premissa em tecnologias de virtualização, como *VirtualBox, LVM, XEN, VMWare, VirtualPC*, etc. Nosso modelo assume que o sistema *host* é inacessível externamente, o que também pode ser garantido através do uso do modo *bridge* e/ou *firewalls* no sistema hospedeiro.

3.3. Tecnologia de virtualização

Dois dos maiores desafios deste trabalho, que são a disponibilização de um serviço confiável de registradores e a garantia do seu isolamento, são resolvidos através do uso de virtualização. Este serviço é implementado nas máquinas hospedeiras que permitem acesso controlado das réplicas. O uso de virtualização traz ainda outros benefícios, como permitir controle e monitoramento da comunicação entre as réplicas e a regeneração de réplicas faltosas, através do uso de imagens do sistema, o que pode ser implementado em trabalhos futuros.

A partir da tecnologia de virtualização é possível criar o serviço confiável de registradores mantendo-o isolado. Diversos gerenciadores de máquinas virtuais (*hypervi-*

sors) possuem mecanismos que podem ser utilizados para garantir o isolamento. No âmbito de testes deste trabalho utilizamos o Hypervisor VirtualBox 4.1.18 e sua funcionalidade de compartilhamento de diretórios afim de simplificar a implementação dos DSR. No entanto, qualquer *hypervisor* tipo 2 poderia ser utilizado e, inclusive, ressaltamos que o uso de diversidade na utilização de *hypervisors* é muito interessante, pois dificulta a exploração de vulnerabilidades por parte de um atacante. A segurança do *hypervisor* é essencial para obter isolamento, por isso alguns trabalhos estudaram como melhorá-la [Murray et al. 2008, Wang and Jiang 2010, Szefer et al. 2011]. Neste artigo, assumimos que um atacante não tem informações suficientes para determinar a arquitetura física-virtual, portanto, outras técnicas podem ser empregadas para garantir o isolamento e a segurança da máquina virtual. Técnicas como Blue Pill [Rutkowska 2006] impedem/dificultam a detecção de máquina virtual de maneira mais contundente.

3.4. Emulação de Memória Compartilhada: Registradores compartilhados distribuídos

Memória compartilhada emulada é uma abstração de registradores de um conjunto de processos que se comunicam através de troca de mensagens [Guerraoui and Rodrigues 2006]. Esta definição é realmente atraente, já que permite que a memória compartilhada seja construída utilizando-se qualquer tecnologia para compartilhamento de memória. Um memória compartilhada, emulada ou não, pode ser vista como um *array* de registradores compartilhados. Consideramos aqui a definição sob a ótica do programador. O tipo de registrador compartilhado especifica que operações podem ser efetuadas e os valores retornados por elas [Guerraoui and Rodrigues 2006]. Os tipos mais comuns são registradores de leitura/escrita. As operações de um registrador são invocadas pelos processos do sistema para troca de informações.

Neste trabalho criamos os registradores compartilhados distribuídos (DSR - acrônimo para *Distributed Shared Registers*), isto é, uma memória compartilhada emulada baseada em troca de mensagens, desenvolvida sobre uma rede controlada e por meio do uso de diretório compartilhados, com controle de acesso. Nós assumimos que a rede controlada só é acessada por componentes dos DSRs. Os DSRs são implementados nos *hosts* das máquinas virtuais dos sistemas e assume-se que o *hypervisor* assegura o isolamento entre *guests* e *hosts*. Os DSR são utilizados para implementar um serviço de *atomic multicast* que garante que todos os servidores corretos irão entregar as mesmas mensagens, em uma mesma ordem e, se o remetente é correto, todos os servidores irão garantir a entrega da mensagem enviada. Os DSRs são capazes de efetuar apenas duas operações (1) $read()$, que lê a última mensagem escrita nos DSR e (2) $write(m)$, que escreve a mensagem m nos DSR. Assume-se duas propriedades com relação às operações acima: (i) progressão (*liveness*) - a operação eventualmente termina; (ii) segurança (*safety*) - a operação de leitura sempre retorna o último valor escrito.

Para garantirmos isto, cada servidor possui um arquivo em que seu *guest* tem acesso para escrita e um segundo arquivo onde o mesmo possui acesso apenas para leitura. Todos os acessos são feitos por um único processo [Guerraoui and Rodrigues 2006]. O primeiro arquivo é espaço da réplica e nenhuma outra réplica pode escrever nele. O segundo arquivo representa o espaço de registradores das demais réplicas e é atualizado pelos DSR. O *hypervisor* fornece suporte para fazer com que um arquivo criado no *host* esteja acessível no *guest*, forçando as permissões de escrita e leitura.

Os DSR aceitam apenas mensagens tipadas, e existem apenas três tipos: (i) PRO-

POSE, (ii) ACCEPT e (iii) CHANGE. As mensagens sem tipo ou com tipos não permitidos são ignoradas. Todas as mensagens trocadas nos DSR são automaticamente identificadas, isto significa que, quando um servidor escreve uma mensagem, os DSR colocam o ID da réplica na mensagem, de modo que uma réplica não possa se passar por outra. Assumimos que a comunicação é feita através de *fair links* com as seguintes propriedades: se o remetente e o destinatário de uma mensagem são ambos corretos, então (1) se a mensagem é enviada infinitamente para um destinatário correto, então ela é recebida infinitas vezes; (2) existe um tempo T, tal que, se a mensagem é retransmitida infinitas vezes para um destinatário correto de acordo com um tempo t_0, então o destinatário recebe a mensagem pelo menos uma vez antes de $t_0 + T$; e (3) as mensagens não são modificadas no canal [Yin et al. 2003]. Esta suposição parece razoável na prática, uma vez que os DSR são síncronos e podem ser afetados apenas por faltas de *crash*, baseado no isolamento proporcionado pelo *hypervisor*.

4. Algoritmo

4.1. Visão geral do protocolo

O protocolo necessita que apenas um servidor seja líder, cuja responsabilidade é propor ordem de execução para as requisições de clientes. Os demais servidores são apenas réplicas do serviço. A escolha do primeiro líder é baseada em configuração. A mudança de liderança ocorre sempre que a maioria das réplicas ($f + 1$) concordam ser necessário. O protocolo se inicia quando um cliente requisita a execução de alguma tarefa nos servidores. O líder é responsável por ordenar e difundir mensagens dos clientes. A ordenação acontece quando o líder escreve nos DSR uma mensagem do tipo PROPOSE. Mensagens deste tipo incluem a mensagem original do cliente seguida de um número de ordenação para a mesma. Cada servidor delibera individualmente se deve ou não aceitar a proposta. Aceitar a proposta significa escrever nos DSR uma mensagem do tipo ACCEPT, esperar por $f - 1$ mensagens de aceitação (pois já possui a sua própria e a mensagem de PROPOSE, isto é $f - 1 + 2 \implies f + 1$), executar a tarefa na ordem estipulada e responder para o cliente com o resultado.

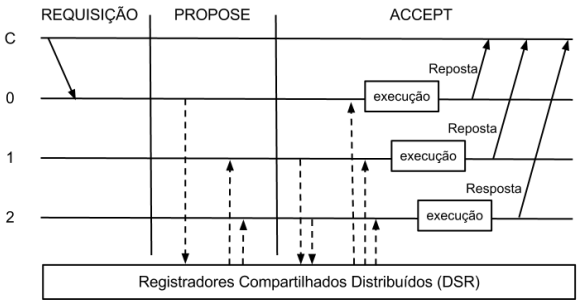


Figura 2. Fluxo da requisição à resposta

Como em outros sistemas tolerantes a faltas bizantinas [Kotla et al. 2008, Correia et al. 2004, Yin et al. 2003, Veronese et al. 2011, Castro and Liskov 2002], para lidar com o problema de um servidor malicioso s_j, que poderia deixar de difundir mensagens, o cliente espera por pelo menos $f + 1$ respostas de diferentes servidores por um tempo $T_{reenviar}$. Após $T_{reenviar}$ o cliente reenvia a requisição para todos os servidores. Uma réplica honesta (não líder) ao receber uma requisição correta, solicita uma

mudança de liderança. Se $f + 1$ servidores corretos solicitarem a mudança de líder, então a mudança ocorre e o protocolo progride. Entretanto, o sistema de *payload* é assumidamente assíncrono, portanto, não existem limitantes temporais para sua comunicação, não sendo possível definir um valor $T_{reenviar}$ "ideal". Em [Correia et al. 2004] é mostrado que o valor $T_{reenviar}$ envolve uma troca: se for alto demais, o cliente pode esperar tempo demais para ter sua operação executada; se for baixo demais, o cliente pode fazer o reenvio do comando sem necessidade. O valor deve ser selecionado considerando-se essa troca. Se o comando é reenviado sem necessidade, as duplicatas são descartadas pelo sistema.

Esta seção oferece uma descrição mais profunda do algoritmo. A sequência de operações do algoritmo é apresentada em um nodo líder e em um nodo não líder. Primeiramente considera-se a operação em caso normal (sem a presença de faltas) e posteriormente a operação sob a presença de faltas. O diagrama de fluxo da operação em caso normal pode ser visto na figura 2.

4.2. Operação em caso normal

1. O protocolo inicia-se quando um cliente c envia ao servidor líder a mensagem $\langle REQUEST, o, t, c_a, v\rangle_{\sigma ci}$ com a operação o que deseja que seja executada e o seu endereço c_a. O campo t é o *timestamp* da requisição para assegurar a semântica de apenas uma execução, em outras palavras, os servidores não executam uma operação do cliente c cujo *timestamp* não seja maior que último *timestamp* para o mesmo cliente. Esta política impede que uma mesma tarefa seja executada inúmeras vezes. O campo v é o vetor que armazena os MACs gerados pelo cliente para cada servidor. Os MACs são gerados utilizando-se a mensagem do cliente e as chaves que foram compartilhadas previamente entre o cliente e os servidores. Em função disto, cada servidor pode averiguar a integridade da mensagem, descartando-a se necessário.

Constants:
f : int // Maximum tolerated faults
T : int // Maximum waiting time for a proposal to be decided
Variables:
accepted : int // counter of acceptance for some ordering
upon receive $\langle REQUEST, o, t_j, c_a, v\rangle_{\sigma ci}$ from client
if $t_j \leq t_{j-1}$ *for* c_i *OR isWrong(* v *)* **then**
    return;
end
write($\langle PROPOSE, n, \langle REQUEST, o, t_j, c_a, v\rangle_{\sigma ci}\rangle_{\sigma si}$) into DSR;
accepted = waitForAcceptance(T);
if *accepted* $\geq f + 1$ **then**
    store $\langle PROPOSE, n, \langle REQUEST, o, t_j, c_a, v\rangle_{\sigma ci}\rangle_{\sigma si}$ in the execution buffer;
end
return;

Algoritmo 1. Algoritmo executado pelo líder.

2. Após receber a requisição do cliente e verificar sua integridade usando o MAC em v, o líder gera uma proposta e a escreve nos DSR. A mensagem $\langle PROPOSE, n_m, m\rangle_{\sigma s}$ gerada pelo líder possui em seu corpo m, que é a mensagem original do cliente, e n_m, que representa o número de ordenação para a mensagem. Vale ressaltar que, como foi discutido, os DSR automaticamente adicionam nas mensagens o identificador do servidor que as registrou. Após escrever a mensagem, o líder aguarda pelas mensagens de aceite de acordo com a proposta. Em paralelo, outras propostas podem ser ordenadas, a medida que chegam requisições.

Após receber f mensagens concordando com a proposta (pois a proposta mais f aceites constituem as $f + 1$ mensagens mínimas para o acordo) o líder guarda a mensagem e a executa na ordem estipulada. O resultado da execução é enviado para o cliente em uma mensagem $\langle REPLY, t, c_a, r_a, r \rangle_{\sigma si}$ que contém o endereço do cliente c_a, o *timestamp* t, o endereço da réplica r_a, e o resultado r. Este comportamento pode ser visto no algoritmo 1. Como foi discutido em 3., todas as mensagens trocadas nos DSR são eventualmente entregues se nem o destinatário, nem o remetente, tiverem sofrido falta de *crash*.

Constants:
f : int // Maximum tolerated faults
T : int // Maximum waiting time for a proposal to be decided.
Variables:
accepted : int // counter of acceptance for some ordering
upon read $\langle PROPOSE, n, \langle REQUEST, o, t, c_a, v \rangle_{\sigma ci} \rangle_{\sigma ss}$ from DSR
if *isValid(v) and isValid(n) and* $t_j > t_{j-1}$ *for* c_i **then**
    write($\langle ACCEPT, n, h_m \rangle_{\sigma si}$) into DSR;
    accepted = waitForAcceptance(T);
    if *accepted* $\geq f + 1$ **then**
        store $\langle PROPOSE, n, \langle REQUEST, o, t_j, c_a, v \rangle_{\sigma ci} \rangle_{\sigma si}$ in the execution buffer;
    end
else
    write($\langle CHANGE, h_m, s_s \rangle_{\sigma si}$) into DSR and bufferize;
end
return;

Algoritmo 2. Algoritmo executado pelo(s) não líder(es).

3. Sempre que a réplica r_j recebe uma proposta do líder, r_j a valida, o que significa que (i) é feita a verificação da integridade da mensagem do cliente usando o MAC em v, além da verificação de seu $timestamp$ e (ii) é feita a verificação de que nenhuma outra mensagem tenha sido ordenada com o mesmo n_m. Após serem feitas as verificações, se r_j aceitar a proposta, então é criada uma mensagem $\langle ACCEPT, n_m, h_m \rangle_{\sigma sj}$ que é escrita nos registradores. Os campos da mensagem representam o *hash* da mensagem do cliente h_m e o número de ordenação proposto pelo líder n_m. Após a escrita nos registradores, a réplica aguarda por $f - 1$ mensagens de aceitação (já que já possui dois aceites, a sua mensagem e a proposta do líder). Em paralelo, outras mensagens podem ser aceitas, à medida que propostas cheguem aos DSR. A réplica r_j executa as requisições na ordem em que foram aceitas e responde aos clientes com mensagens de REPLY. Este comportamento pode ser observado no algoritmo 2.
4. Após receber $f + 1$ respostas iguas entre si e de diferentes servidores, o cliente finalmente as aceita como corretas.

4.3. Operação sob a presença de faltas

A operação sob a presença de faltas implica na mudança de líder, portanto, fazemos uma breve explicação de como essa mudança se desenvolve.

Mudança de líder: durante a configuração do sistema, todas as réplicas recebem um número de identificação. Estes números são sequenciais e iniciados em zero. Todas as réplicas conhecem o identificador do líder L_{id} e o número total de réplicas no sistema N. Quando $f + 1$ réplicas corretas suspeitam do líder atual, elas simplesmente definem $L_{id} = L_{id} + 1$ como o próximo líder se $L_{id} < N$, senão, $L_{id} = 0$

Como foi discutido, a réplica r_j valida qualquer proposta que receba. Se por alguma razão a réplica decidir não aceitar a proposta ou se r_j receber uma mensagem correta de

um cliente e verificar que a mesma não foi proposta, r_j solicitará uma mudança de líder. O diagrama de fluxos da figura 3 exemplifica um dos casos de execução sob a presença de faltas.

1. O protocolo pode iniciar o modo faltoso de acordo com as duas maneiras abaixo:
   (a) Quando a réplica r_j recebe uma mensagem do tipo CHANGE, mas ainda não suspeita do líder, ela armazena a mensagem em seu *buffer* para utilizá-la futuramente, se necessário.
   (b) Quando a réplica r_j suspeita do líder por alguma mensagem m, então r_j escreve nos DSR e em seu *buffer* uma mensagem $\langle CHANGE, h_m, s_s \rangle_{\sigma sj}$ que contém o *hash* da mensagem h_m e o id s_s do líder do qual r_j suspeita.
2. A réplica r_j inicia uma busca em seu *buffer* de suspeita para encontrar $f+1$ mensagens relacionadas à mensagem m. Caso r_j encontre $f+1$ (incluindo ao seu próprio) identificadores para a mesma mensagem, então a réplica efetua a mudança de líder.
3. Se o identificador do líder for o da réplica r_j, então a réplica recomeça o processo de ordenação utlizando-se das mensagens de aceite escritas nos DSR para definir o estado atual do sistema. Recomeçando o processo de acordo com a última mensagem que foi aceita pela maioria. Se o identificador não for o da réplica r_J, então a réplica aguardará pela mensagem do novo líder.
4. Quando r_j receber uma proposta do novo líder, então ela limpará seu *buffer* e voltará o protocolo para a operação sem a presença de faltas.

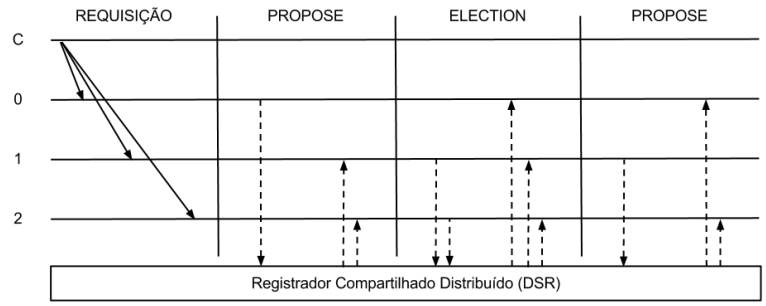
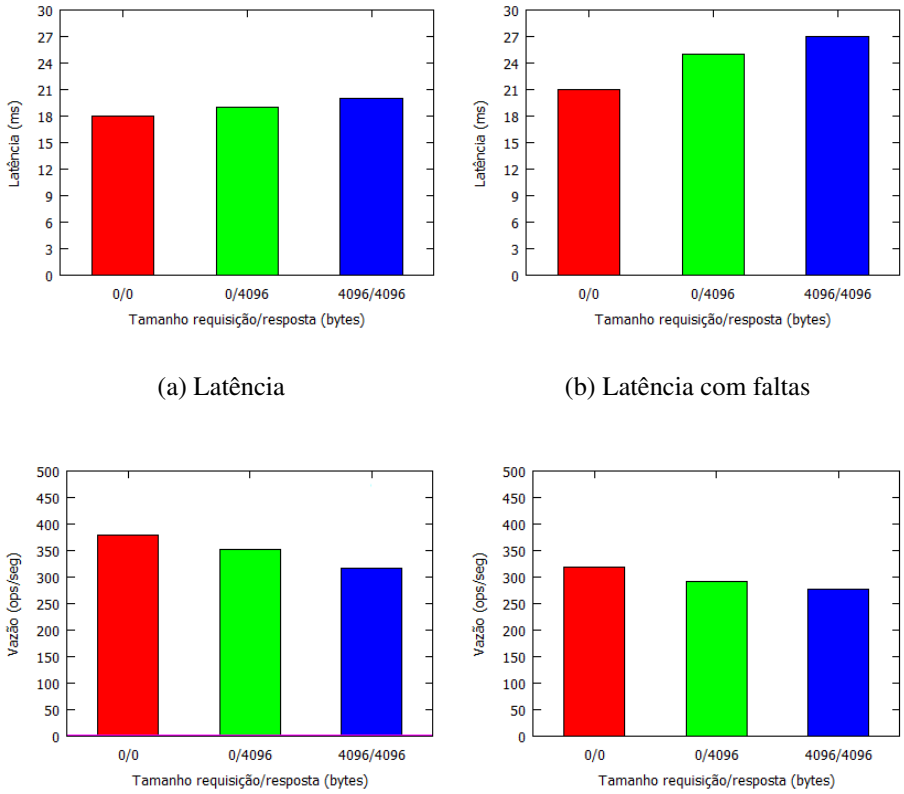


Figura 3. Fluxo de uma mudança de líder

5. Implementação e Avaliação de Desempenho

Os algoritmos foram implementados usando a linguagem Java, JDK 1.6.0. Os canais de comunicação foram implementados usando *sockets* TCP da API NIO. Os sistemas operacionais dos *hosts* das máquinas virtuais foram MacOSx Lion e Ubuntu 12.04. Nos *guests* das máquinas virtuais utilizou-se Ubuntu 12.04 e Debian 6 Stable. O *hypervisor* escolhido foi a VirtualBox. As métricas de avaliação escolhidas foram latência e vazão, dado que estas são as métricas largamente usadas para avaliar sistemas computacionais e representam a eficiência do sistema de uma maneira simples [Jian 1991].

Os valores foram obtidos através de *micro-benchmarks* com diferentes cargas. A latência foi obtida pela medida do *round-trip*. Nós medimos o tempo entre o envio e o recebimento de um grupo de mensagens. O raciocínio por trás do uso de *micro-benchmarks* é medir adequadamente o algoritmo sem influências externas. A fim de avaliar a capacidade do protocolo, executou-se com tamanhos diferentes de mensagens. A vazão representa a



(a) Latência

(b) Latência com faltas

(c) vazão em caso normal

(d) vazão com faltas

Figura 4. Desempenho verificado para o RegPaxos.

capacidade do sistema de processamento de mensagens por unidade de tempo, portanto, sendo medido o tempo que demora a receber a resposta de todas as réplicas.

Para avaliar o desempenho do algoritmo na ausência de faltas, executou-se o protocolo em condições normais enviando 10.000 requisições através de um único cliente com três cargas diferentes: 0/0kb, 0/4 kb e 4/4 kb. Com isto temos: uma requisição vazia e uma resposta vazia, uma requisição vazia e uma resposta de 4kb de tamanho e uma requisição de 4kb com uma resposta de 4kb. Para avaliar o algoritmo em um ambiente faltoso, as réplicas foram configuradas para que, quando assumissem o papel de líderes, enviassem uma entre dez respostas incorretas. As figuras 4(a) e 4(b) apresentam a latência para cada carga diferente. A latência foi obtida pela média entre as respostas de todas as requisições. Como podemos observar, a latência tem variações mínimas entre diferentes cargas. A vazão, representada pelas figuras 4(c) e 4(d), foram baseadas no cálculo do tempo total de 10.000 requisições

Dados comparativos entre o RegPaxos e o estado da arte em sistemas BFT são apresentados na tabela 1. Todos os dados consideram execuções na ausência de faltas. Os benefícios do uso do RegPaxos são visíveis quando comparados os números de passos de comunicação e quantidade de réplicas necessária. Nossa abordagem tem a melhor resiliência prática em termos de réplicas, juntamente com [Chun et al. 2007, Correia et al. 2004, Veronese et al. 2011], e também tem o mesmo número de passos que [Kotla et al. 2008, Veronese et al. 2011], mesmo não sendo especulativo. Ao evitar o envolvimento dos clientes, permitimos que um número finito de clientes incorra em faltas.

6. Conclusão

Ao explorar o uso de técnicas de memória compartilhada emulada (registradores compartilhados distribuídos) e virtualização, conseguimos propor uma rede inviolável simples que suporta o nosso algoritmo BFT. Mostrou-se que é possível implementar um algoritmo SMR confiável, com apenas $2f + 1$ réplicas usando tecnologias comuns, como virtualização e abstrações de compartilhamento de dados. Provou-se que é possível criar um modelo híbrido que, mesmo não sendo especulativo, é capaz de atingir o mesmo número de passos de comunicação sem a necessidade de modificar o *software* do servidor e/ou criar componentes de *hardware*. A tecnologia de virtualização é amplamente utilizada e pode proporcionar um bom isolamento entre o serviço confiável e o mundo exterior, e a utilização do DSR simplifica o progresso do protocolo. Além disso, foi possível reduzir o número de passos de comunicação, o que pode reduzir a latência.

Referências

Bessani, A. N., Correia, M., da Silva Fraga, J., and Lung, L. C. (2007). Decoupled quorum-based byzantine-resilient coordination in open distributed systems. In *NCA'07*, pages 231–238.

Castro, M. and Liskov, B. (1998). Practical byzantine fault tolerance. *Operating Systems Review*, 33:173–186.

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., Shenker, S., and Kubiatowicz, J. (2007). Attested append-only memory: Making adversaries stick to their word. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 189–204. ACM.

Correia, M., Neves, N., and Verissimo, P. (2004). How to tolerate half less one byzantine nodes in practical distributed systems. In *Reliable Distributed Systems, 2004. Proceedings of the 23rd IEEE International Symposium on*, pages 174–183. IEEE.

Correia, M., Neves, N., and Verissimo, P. (2012). Bft-to: Intrusion tolerance with less replicas. *The Computer Journal*.

Correia, M., Verissimo, P., and Neves, N. (2002). The design of a cots real-time distributed security kernel. *Dependable Computing EDCC-4*, pages 634–638.

Guerraoui, R. and Rodrigues, L. (2006). *Introduction to reliable distributed programming*. Springer-Verlag New York Inc.

Jian, R. (1991). The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling.

Kotla, R., Clement, A., Wong, E., Alvisi, L., and Dahlin, M. (2008). Zyzzyva: speculative byzantine fault tolerance. *Communications of the ACM*, 51(11):86–95.

Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565.

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

Menezes, A., Van Oorschot, P., and Vanstone, S. (1996). *Handbook of applied cryptography*. CRC.

Murray, D. G., Milos, G., and Hand, S. (2008). Improving xen security through disaggregation. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 151–160. ACM.

Obelheiro, R. R., Bessani, A. N., and Lung, L. C. (2005). Analisando a viabilidade da implementação prática de sistemas tolerantes a intrusões. In *Anais do V Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais - SBSeg 2005*.

Reiser, H. and Kapitza, R. (2007). Vm-fit: supporting intrusion tolerance with virtualisation technology. In *Proceedings of the Workshop on Recent Advances on Intrusion-Tolerant Systems*.

Reiter, M. (1995). The rampart toolkit for building high-integrity services. *Theory and Practice in Distributed Systems*, pages 99–110.

Rodrigues, R., Castro, M., and Liskov, B. (2001). Base: Using abstraction to improve fault tolerance. *ACM SIGOPS Operating Systems Review*, 35(5):15–28.

Rutkowska, J. (2006). Introducing blue pill. *The official blog of the invisiblethings. org*, 22.

Schneider, F. (1982). Synchronization in distributed programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(2):125–148.

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

Stumm, V., Lung, L., Correia, M., da Silva Fraga, J., and Lau, J. (2010). Intrusion tolerant services through virtualization: a shared memory approach. In *Advanced Information Networking and Applications (AINA), 2010 24th IEEE International Conference on*, pages 768–774. IEEE.

Szefer, J., Keller, E., Lee, R. B., and Rexford, J. (2011). Eliminating the hypervisor attack surface for a more secure cloud. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 401–412. ACM.

Veríssimo, P., Neves, N., and Correia, M. (2003). Intrusion-tolerant architectures: Concepts and design. *Architecting Dependable Systems*, pages 3–36.

Veríssimo, P. E. (2006). Travelling through wormholes: a new look at distributed systems models. *SIGACT News*, 37(1):66–81.

Veronese, G., Correia, M., Bessani, A., Lung, L., and Verissimo, P. (2011). Efficient byzantine fault tolerance. *Computers, IEEE Transactions on*, pages 1–1.

Wang, Z. and Jiang, X. (2010). Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *IEEE Symposium on Security and Privacy - SP'10*, pages 380–395. IEEE.

Wangham, M. S., Lung, L. C., Westphall, C. M., and da Silva Fraga, J. (2001). Integrating ssl to the jacoweb security framework: Project and implementation. In *Integrated Network Management'01*, pages 779–792.

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.