DifATo - Difusão Atômica Tolerante a Faltas Bizantinas Baseada em Tecnologia de Virtualização

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. *This paper presents a byzantine fault-tolerant protocol for atomic multicast whose algorithm manages to implement a reliable consensus service with only $2f + 1$ servers to tolerate f faulty ones. For the creation of the algorithm we used common technologies such as virtualization and data sharing abstractions. The system model adopted is hybrid, meaning that the assumptions of synchrony and occurrence of faults consider each component separately. Moreover, in our model, we use two networks to provide the service, a payload network, where messages are exchanged between clients and servers, and a tamperproof network, where the messages are ordered.*

Resumo. *Este trabalho apresenta um protocolo de difusão atômica tolerante a faltas Bizantinas (Byzantine Fault Tolerant - BFT) em que o algoritmo implementa um serviço confiável de consenso com $2f + 1$ servidores tolerando até f faltosos. Para a criação do algoritmo são utilizadas tecnologias comuns como virtualização e abstrações de compartilhamento de dados. O modelo de sistema adotado é híbrido, o que significa que as premissas de sincronismo e ocorrência de faltas consideram cada componente separadamente. Além disso, em nosso modelo, utilizamos duas redes para fornecer o serviço, uma rede de carga, onde são trocadas mensagens entre os clientes e servidores, e uma rede inviolável, onde são feitas as ordenações das mensagens.*

1. Introdução

A dificuldade em construir sistemas distribuídos pode ser drasticamente reduzida através do uso de primitivas de comunicação em grupo, tal como a difusão atômica (ou difusão com ordem total) [Défago et al. 2004]. A difusão atômica assegura que mensagens enviadas para um conjunto de processos serão entregues por estes na mesma ordem, sendo empregada nos mais diversos domínios de aplicação como: sincronização de relógios, CSCW (*Computer Supported Cooperative Work*), memórias distribuídas, replicação de base de dados [Rodrigues et al. 1993, Kemme et al. 2003, Bessani et al. 2006], e é a base para abordagens de replicação de máquina de estados [Schneider 1990], e também o componente principal de muitos sistemas tolerantes a faltas [Castro and Liskov 2002, Yin et al. 2003, Correia et al. 2006, Favarim et al. 2007].

A literatura nos mostra uma quantidade considerável de trabalhos sobre difusão com ordem total, com as mais variadas abordagens e algoritmos. Entretanto, em sua maioria, estes algoritmos consideram modelos de sistema sujeitos apenas a faltas de parada (i.e. *crash*) [Défago et al. 2004, Ekwall et al. 2004], de modo que poucos são os trabalhos que endereçam faltas arbitrárias/Bizantinas [Correia et al. 2006, Reiter 1994]. Em geral, as abordagens usam algoritmos de consenso para estabelecer um acordo acerca da ordenação das mensagens, e necessitam de, pelo menos, $3f + 1$ processos envolvidos no procedimento. Por outro lado, alguns trabalhos propõem a separação do consenso do acordo, o que dá origem a um serviço de consenso [Guerraoui and Schiper 2001, Pieri et al. 2010].

Neste trabalho apresentamos o DifATo (acrônimo para **Dif**usão **A**tômica **To**lerante a Faltas Bizantinas), um serviço de consenso com o propósito de difundir as mensagens atomicamente, a despeito de faltas Bizantinas. O modelo de sistema, bem como a arquitetura que propomos necessita apenas de $2f + 1$ servidores para compor o serviço de consenso e baseia-se em um modelo híbrido, isto é, um modelo onde variam, de componente para componente, as suposições sobre sincronismo e presença/severidade de faltas e falhas [Correia et al. 2002, Veríssimo 2006, Correia et al. 2004]. Em nosso modelo consideramos a existência de uma rede de carga (i.e. *payload*), que é utilizada para a comunicação entre os clientes e o serviço de consenso; e também uma rede inviolável, onde os servidores executam a ordenação das mensagens. Nossa contribuição vem ao encontro da proposição de melhorias no serviço de consenso, a fim de torná-lo tolerante a faltas Bizantinas e com um custo mais reduzido (i.e. com apenas $2f + 1$ servidores).

O restante deste artigo está organizado da seguinte forma. A Seção 2 apresenta os trabalhos relacionados; a Seção 3 descreve o modelo de sistema adotado para este trabalho; na Seção 4 são apresentados os algoritmos que compõem a proposta, e uma breve descrição das provas para os algoritmos são apresentadas na Seção 5. Na Seção 6 se apresenta alguns aspectos de implementação e os resultados obtidos, e por fim, a Seção 7 conclui o artigo.

2. Trabalhos Relacionados

O problema de difusão atômica tem sido amplamente estudado nas últimas décadas [Rodrigues et al. 1993, Reiter 1994, Veríssimo 2006, Kemme et al. 2003, Ekwall et al. 2004, Correia et al. 2006]. Em sua maioria, as abordagens consideram que os processos podem sofrer apenas por faltas de parada, isto é, não considerando para tanto, faltas de natureza arbitrária ou Bizantina [Lamport et al. 1982].

Em [Reiter 1994] é apresentado o Rampart, um suporte para difusão atômica confiável em sistemas sujeitos a faltas Bizantinas. O algoritmo deste é baseado em um serviço de associação a grupo, requerendo que pelo menos um terço de todos os processos pertencentes a visão atual entrem em acordo sobre a exclusão de alguns processos do grupo. A difusão atômica é feita por um membro do grupo chamado de sequenciador, cuja responsabilidade é determinar a ordem para as mensagens enviadas na visão atual. Na próxima visão, outro sequenciador é escolhido por um algoritmo determinístico. O Rampart assume um modelo de sistema assíncrono, com canais FIFO confiáveis, e uma infraestrutura de chaves públicas conhecida por todos os processos. Com a suposição de canais de comunicação autenticados, a integridade das mensagens trocadas entre dois processos não Bizantinos é garantida.

Guerraoui e Schiper propuseram um serviço genérico de consenso [Guerraoui and Schiper 2001] para resolver problemas de acordo, dentre os quais

está incluída a difusão atômica. Este serviço constitui a base para nossa proposta. Neste caso, o modelo proposto pelos autores considera um ambiente sujeito a apenas falhas por parada, o qual possui um serviço de consenso que separa o consenso do problema de acordo a ser resolvido. O sistema requer que se tenha um detector de falhas perfeito [Chandra and Toueg 1996] (baseando o serviço de consenso) e a resiliência varia de acordo com o desempenho desejado.

Alguns anos mais tarde, Correia e Veríssimo mostraram uma transformação de consenso para difusão atômica [Correia et al. 2006], em que o modelo de sistema apresentado assumia um ambiente Bizantino, no qual $f = \lfloor \frac{(n-1)}{3} \rfloor$ faltas eram toleradas. Os autores implementaram um protocolo de consenso multivalorado sobre um consenso binário aleatório, e um protocolo de difusão confiável. O protocolo de difusão atômica é criado através de sucessivas transformações a partir do protocolo de consenso. A difusão atômica é feita através do uso de um vetor de *hashes*. Cada processo do sistema propõe um valor para o vetor de consenso (que é o vetor com os *hashes* das mensagens). O protocolo de vetor de consenso decide sobre um vetor X_i com pelo menos $2f + 1$ vetores H de diferentes processos. Em seguida, as mensagens são armazenadas em um conjunto para serem atomicamente entregues na ordem pré-estabelecida.

Mais recentemente, Pieri et al. propôs uma extensão ao serviço genérico de consenso para ambientes Bizantinos [Pieri et al. 2010]. O modelo de sistema proposto possuía $n_c = 3f_c + 1$ clientes e $n_s = 2f_s + 1$ servidores, e fazia uso de máquinas virtuais para prover o serviço genérico de consenso. Na proposta dos autores, o consenso atômico se inicia sempre que um dos processos, conhecido no protocolo por iniciador, difunde de maneira confiável uma mensagem m_i para o conjunto de clientes. Ao receber a mensagem m_i, cada cliente envia uma proposta de ordenação de m_i para o serviço genérico de consenso. Quando os servidores recebem $n_c - f_c$ propostas de clientes para a mesma instância de consenso, cada servidor inicia o protocolo de consenso, e então o resultado do protocolo é enviado aos clientes. A importância deste trabalho para a literatura, é que ele foi o primeiro a tornar o serviço genérico de consenso disponível para ambientes sujeitos a faltas Bizantinas, a despeito do número de clientes ser limitado. Além disso, o sistema precisava lidar com o a existência de clientes faltosos, diminuindo assim, a resiliência do sistema. De um ponto de vista, isto é aceitável quando se pretende trabalhar com problemas genéricos de acordo, porém, em se tratando especificamente do problema de difusão atômica, é algo que se torna custoso.

3. Modelo de Sistema e Arquitetura

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 compõem 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 de consenso. Cada servidor possui uma máquina virtual que contém apenas um sistema como *guest*. O modelo de falhas admite que um número finito de clientes pode sofrer faltas por parada, e até $f \leq [\frac{n-1}{2}]$ servidores podem falhar em 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. Todavia, assumimos a independência de falhas, de tal maneira que a ocorrência de uma falta em um determinado servidor, é independente da ocorrência da mesma falta em outro servidor. Na prática, isto é possível por meio do uso extensivo de diversidade (diferentes *hardware/software*, sistemas operacionais, máquinas virtuais, bases de dados, linguagens de programação, etc) [Obelheiro et al. 2005].

Nosso modelo de sistema prevê o uso de duas redes de comunicação. A primeira, que é a rede de carga (ou *payload*) é assíncrona e é utilizada para transferência de dados da aplicação, isto é, para a interação entre os clientes e servidores. Assim, não fazemos quaisquer suposições baseadas em tempo para a rede de carga, de modo que sua utilização ocorre apenas para envio de requisições e respostas entre clientes e servidores. Por outro lado, a segunda rede, que implementa um serviço de Registradores Compartilhados Distribuídos (i.e. uma memória compartilhada) é controlada, e é utilizada apenas para a interação entre os servidores, para que estes possam trocar as mensagens do protocolo de consenso. Deste modo, para esta rede assumimos as seguintes hipóteses:

- possui um número finito e conhecido de membros;
- é segura e resistente a qualquer possível ataque, e pode falhar apenas por parada (*crash*);
- é capaz de executar operações com delimitação temporal;
- provê apenas duas operações, uma para leitura e outra para escrita, em registradores; estas operações não podem ser afetadas por faltas maliciosas.

Cada máquina física possui seu próprio espaço dentro dos registradores compartilhados distribuídos, e é neste espaço que cada máquina virtual registra as mensagens do tipo PROPOSE, ACCEPT e CHANGE. Todos os servidores podem escrever em todo o espaço de registradores, independente de quem tem o direito de escrita no mesmo.

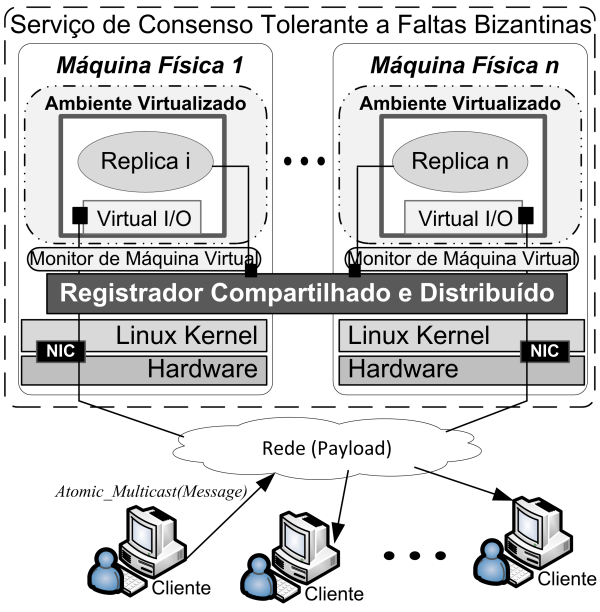


Figura 1. Visão geral da arquitetura.

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 será recebida em

algum momento 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]. Para isto, empregamos o uso de códigos de autenticação de mensagens (MACs) como *checksums* criptográficos, o que é viabilizado apenas pelo uso de criptografia simétrica [Menezes et al. 1996, Castro and Liskov 2002]. Não obstante, é requerido o compartilhamento de chaves simétricas pelos processos, a fim de permitir o uso de MACs, onde assumimos que estas chaves são distribuídas antes do protocolo ser executado. Em termos de implementação, isto pode ser resolvido usando protocolos de distribuição de chaves disponíveis na literatura [Menezes et al. 1996]. Todavia, salientamos que este problema está fora do escopo deste trabalho.

Assumimos que apenas as máquinas físicas podem, de fato, conectar-se a rede controlada usada pelos registradores. Isto significa que os registradores são acessíveis apenas pelas máquinas físicas que compõe o sistema e hospedam máquinas virtuais. Com isso, o acesso aos registradores não é possível através do acesso às máquinas virtuais. Cada processo é encapsulado em sua própria máquina virtual, de modo a assegurar o isolamento. Toda a comunicação cliente-servidor acontece em uma rede separada (rede de carga) e, do ponto de vista dos clientes, a máquina virtual é transparente. Portanto, os clientes não são capazes de identificar a arquitetura físico-virtual. Cada máquina possui apenas uma placa de interface de rede (NIC), o hospedeiro utiliza *firewall* e/ou modo *bridge* para assegurar a divisão das redes. Assumimos que as vulnerabilidades do hospedeiro não podem ser exploradas através da máquina virtual. O monitor da máquina virtual assegura o isolamento, garantindo que um atacante não tem meios para acessar o hospedeiro através da máquina virtual. Esta é uma característica presente em grande parte das tecnologias de virtualização mais comuns, tal como VirtualBox, LVM, XEN, VMWare, VirtualPC, etc. Nosso modelo assume que o sistema hospedeiro é inacessível externamente, o que é também garantido pelo uso do modo *bridge* e/ou *firewall* no sistema hospedeiro.

3.1. Registradores Compartilhados Distribuídos (RCD)

A memória compartilhada emulada consiste em uma abstração de registradores disponível para um conjunto de processos, na qual a comunicação subjacente é realizada através de troca de mensagens [Guerraoui and Rodrigues 2006]. Esta definição é realmente atraente, pois permite que a memória compartilhada seja construída utilizando qualquer tecnologia para compartilhamento de memória. A memória compartilhada, emulada ou não, pode ser vista como um *array* de registradores compartilhados, em que consideramos a definição sob o ponto de vista do programador. O tipo do registrador compartilhado especifica quais operações podem ser realizadas e os valores retornados pela operação [Guerraoui and Rodrigues 2006]. Os tipos mais comuns são os de leitura/escrita. As operações dos registradores são invocadas pelos processos do sistema para troca de informações. Para a realização deste trabalho, criamos uma abstração de memória emulada compartilhada, que denominamos como Registradores Compartilhados Distribuídos (RCD). Esta abstração é baseada em troca de mensagens através de uma rede controlada, que faz uso de arquivos locais para efetuar as operações. Assumimos que a rede controlada é acessível apenas por componentes do RCD. Os registradores são implementados nos hospedeiros das máquinas virtuais, onde assumimos que o monitor da máquina virtual assegura o isolamento entre eles e seus convidados.

O acesso aos RCDs é realizado por meio de apenas duas operações:

1. $read()$ - Usada para ler a última mensagem escrita nos RCDs;

2. $write(m)$ - Usada para escrever a mensagem m nos RCDs.

A execução destas operações leva em consideração duas propriedades básicas, que são:

(i) Vivacidade (*liveness*) - A operação eventualmente termina;
(ii) Segurança (*safety*) - A operação de leitura sempre retorna o último valor escrito.

Em termos de implementação, para tornar possível o atendimento destas propriedades, em cada servidor é criado um arquivo onde o convidado tem acesso apenas para escrita e outro onde o convidado tem acesso apenas para leitura, e todos os acessos são feitos por um único processo [Guerraoui and Rodrigues 2006]. Os RCDs aceitam apenas mensagens que estejam de acordo com a especificação do protocolo, isto é, tipificadas em que se admite apenas os tipos: (*i*) PROPOSE, (*ii*) ACCEPT e (*iii*) CHANGE. Com isso, todas as mensagens que não seguem esta especificação são ignoradas e descartadas.

De outro modo, assumimos que a comunicação nos RCDs se dá através de canais do tipo *fair links*, os quais atendem as condições de que, se ambos o remetente e o destinatário de uma mensagem são corretos, então [Yin et al. 2003]:

1. Se uma mensagem for enviada infinitas vezes para um destinatário, então a mensagem é recebida infinitas vezes;
2. Existe um atraso T de modo que, se uma mensagem é retransmitida infinitas vezes para um destinatário a partir de um tempo t_0, então o destinatário receberá a mensagem pelo menos uma vez antes de $t_0 + T$;
3. As mensagens não são modificadas no canal.

Entendemos que estas suposições são bastante razoáveis na prática, já que os RCDs são implementados em uma rede síncrona e separada, e que sofre apenas faltas por parada - baseado no isolamento provido pelo monitor de máquinas virtuais.

3.2. Propriedades da Difusão Atômica

O problema de difusão atômica, ou difusão confiável com ordem total, consiste em garantir a entrega de um conjunto de mensagens, na mesma ordem, para todos os processos que fazem parte de um sistema. A definição em um contexto Bizantino pode ser feita, considerando as seguintes propriedades:

DA1 *Validade* - Se um processo correto difunde uma mensagem m, então algum processo correto eventualmente entrega m.
DA2 *Acordo* - Se um processo correto entrega uma mensagem m, então todos os processos corretos eventualmente entregam m.
DA3 *Integridade* - Para qualquer mensagem m, todo processo correto entrega m no máximo uma vez, e se o remetente de m for correto, então m foi anteriormente difundida por este remetente.
DA4 *Ordem total* - Se dois processos corretos entregam duas mensagens com os prefixos m_{i-1} e m_i, então ambos os processos entregam as duas mensagens de maneira que m_{i-1} antecede m_i.

4. Algoritmo DifATo

Discussão: O procedimento necessita que apenas um servidor atue como sequenciador. Este servidor é responsável por propor as ordens para as mensagens dos clientes. Os demais servidores são apenas réplicas do serviço. Inicialmente o sequenciador é o processo com o menor número identificador (zero). A mudança de sequenciador acontece sempre que a maioria dos servidores ($f+1$) concordarem que esta condição é necessária. Como em outros sistemas tolerantes a faltas Bizantinas [Correia et al. 2004, Castro and Liskov 2002, Yin et al. 2003], é necessário lidar com o problema de um servidor p_j malicioso que pode descartar mensagens dos clientes. Em função disto, os clientes enviam suas mensagens para serem ordenadas, e esperam recebê-las de volta devidamente ordenadas, em até um tempo $T_{reenviar}$. Depois de passado este tempo, o cliente envia sua mensagem para todos os servidores. Um servidor correto, quando recebe uma mensagem do cliente e não é um sequenciador, solicita uma mudança de sequenciador. Se $f + 1$ servidores solicitam uma mudança de sequenciador, então os servidores corretos efetuam a mudança e o protocolo prossegue. Entretanto, a rede de carga é assumidamente assíncrona, por isso, não existem limites para os atrasos na comunicação, e não é possível definir um valor ideal para $T_{reenviar}$. Correia [Correia et al. 2004] mostra que o valor de $T_{reenviar}$ envolve uma troca: se o valor for muito alto, o cliente pode esperar demais pela ordenação da mensagem; se baixo demais, o cliente pode reenviar a mensagem sem necessidade. O valor deve considerar essa troca. Se a mensagem é reenviada sem necessidade, sua duplicata é descartada pelo sistema.

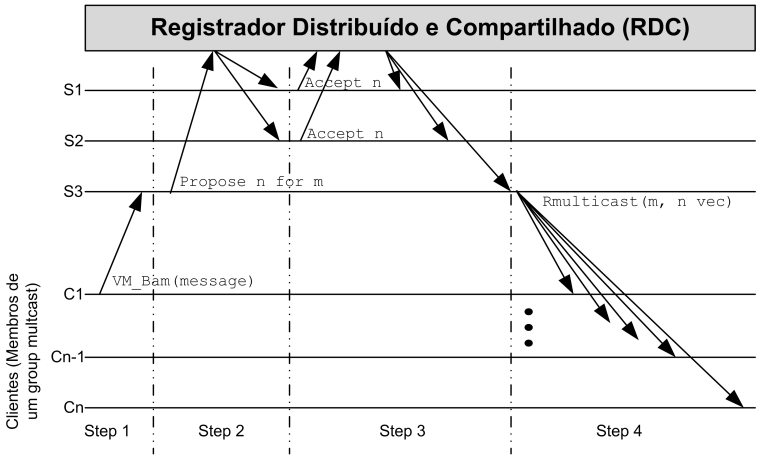


Figura 2. Fluxo da difusão atômica.

Esta seção oferece uma descrição mais aprofundada do algoritmo. A sequência de operações dos algoritmos é apresentada tanto no nó que atua como sequenciador, como naqueles que não desempenham este papel, isto é, os demais nós servidores. Primeiramente é considerada a operação do algoritmo com a ausência de faltas (i.e. caso normal) e, na sequencia, com a presença de faltas. O diagrama de fluxos da operação na ausência de faltas pode ser visto na Figura 2. Por clareza de apresentação, consideramos apenas um único grupo de difusão.

4.1. Operação em Caso Normal

Passo 1) O procedimento inicia-se quando algum cliente c_i envia a mensagem $\langle ORDER, m, t, v\rangle_{\sigma ci}$ para o sequenciador com sua mensagem m incluída. O

campo t é a marca de tempo da mensagem para assegurar a semântica de apenas uma ordenação por mensagem. Desta maneira os servidores só executam a ordenação de mensagens cuja marca de tempo seja maior que a anterior, para um mesmo cliente. O campo v é o vetor que gera um MAC por servidor, cada um obtido através da chave compartilhada entre clientes e servidores. Portanto, cada servidor pode testar a integridade da mensagem utilizando este vetor. Caso uma mensagem já tenha sido ordenada, o servidor apenas a reenvia para o cliente.

Algoritmo 1: Algoritmo executado pelo nó sequenciador.

```
   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
 1 upon receive ⟨ORDER, m, t_j, v⟩_σci from client
 2 if t_j ≤ t_{j-1} for c_i then
 3 |   if has(m) into buffer then
 4 |   |   rmulticast( getOrdered( m ) from buffer );
 5 |   end
 6 |   return;
 7 end
 8 if isWrong( v ) then
 9 |   return;
10 end
11 write( ⟨PROPOSE, n_o, ⟨ORDER, m, t_j, v⟩_σci, mac⟩_σsi ) into RCD;
12 accepted = waitForAcceptance( T );
13 if accepted ≥ f + 1 then
14 |   store ⟨PROPOSE, n_o, ⟨ORDER, m, t_j, v⟩_σci, mac⟩_σsi in the atomic buffer;
15 end
16 return;
```

Passo 2) Depois de verificar se o MAC em v é correto e se a marca de tempo é válida para a mensagem do cliente, o sequenciador gera uma mensagem $\langle PROPOSE, n_o, o, mac\rangle_{\sigma si}$ onde o representa a mensagem original do cliente, n_o é o número de ordenação para o e mac é o MAC gerado pelo sequenciador. Os RCDs automaticamente identificam a mensagem proposta com o id do sequenciador. O sequenciador espera pela aceitação dos demais servidores, isto é, f processos concordando com a proposta. Ao ter a mensagem aceita, o sequenciador salva a mensagem e a ordenação em seu *buffer*. Este comportamento pode ser observado no algoritmo 1. Como foi discutido, todas as mensagens escritas nos RCDs serão entregues se o destinatário e o remetente não sofreram uma parada (*crash*).

Passo 3) Ao receber uma proposta, o servidor s_k a valida: (i) s_k verifica, usando o vetor de MACs, se o conteúdo da mensagem m está correto e (ii) verifica se não existe outra proposta anteriormente aceita para o número de ordenação n. Depois de aceitar a proposta, s_k escreve uma mensagem $\langle ACCEPT, n_o, h_m, mac\rangle_{\sigma sk}$ nos registradores. Esta mensagem possui o *hash* da mensagem do cliente h_m, o número de ordenação aceito e o MAC mac gerado pelo servidor. Após escrever a mensagem de aceite, o processo aguarda por $f - 1$ mensagens de aceitação para, então, salvar a mensagem no *buffer*. Este comportamento pode ser visto no algoritmo 2.

Passo 4) O sequenciador difunde de maneira confiável a mensagem com o número de ordenação e um vetor de MACs assinado por pelo menos $f + 1$ servidores diferentes que aceitaram a ordem proposta. Após receber e validar o vetor, os clientes finalmente aceitam a mensagem e a entregam na ordem estipulada.

Algoritmo 2: Algoritmo executado pelo(s) nó(s) não sequenciador(es).

```
   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
1  upon read ⟨PROPOSE, n_o, ⟨ORDER, m, t_j, v⟩_σci, mac⟩_σss from RCD
2  if isValid( v ) and isValid( n ) and t_j > t_j−1 for c_i then
3      write( ⟨ACCEPT, n_o, h_m⟩_σsi ) into RCD;
4      accepted = waitForAcceptance( T );
5      if accepted ≥ f + 1 then
6          store ⟨PROPOSE, n_o, ⟨ORDER, m, t_j, v⟩_σci, mac⟩_σss in the atomic buffer;
7      end
8  else
9      write( ⟨CHANGE, h_m, s_s⟩_σsi ) into RCD and bufferize;
10 end
11 return;
```

4.2. Operação em Situação com Faltas

A operação na presença de faltas implica que uma mudança de sequenciador ocorrerá, portanto, faremos uma breve explicação de como isso se desenvolve.

Mudança de sequenciador: Durante a configuração do sistema, todos os servidores recebem um número de identificação. Estes números são sequenciais e iniciam em zero. Todos os servidores conhecem o identificador S do sequenciador e o número total de servidores no sistema. Quando $f + 1$ servidores corretos suspeitam do sequenciador atual, eles simplesmente definem $S = S + 1$ como o próximo sequenciador, se $S < n - 1$, senão $S = 0$.

Ao validar uma proposta, o servidor s_k verifica, usando o MAC no vetor v, se o conteúdo da mensagem está correto. Se o conteúdo estiver correto e se o número de ordenação estiverem corretos, o servidor aceita a proposta. Caso contrário, s_k vai solicitar uma mudança de sequenciador:

1. Um servidor correto pode entrar em modo faltoso de operação de duas maneiras:
   (a) Quando o servidor s_k lê dos registradores uma mensagem de mudança de sequenciador, mas ainda não suspeita do servidor s_s. O servidor apenas armazena a mensagem em seu *buffer* local para utilização futura.
   (b) Se o processo s_k suspeita do sequenciador s_s com relação à mensagem m, então s_k escreve uma mensagem $\langle CHANGE, sid, h_m, s_s \rangle_{\sigma_{sk}}$ nos RCDs contendo o sid como seu próprio identificador, o *hash* da mensagem h_m que originou a suspeita e o identificador s_s do servidor em suspeita.
2. O servidor s_k inicia uma busca em seu *buffer* local, no intuito de encontrar $f + 1$ mensagens de mudança relacionadas à mensagem m e ao servidor s_s. Caso s_k encontre $f + 1$ (incluindo o próprio s_k) diferentes sid para a mesma mensagem, então o servidor efetua a mudança de sequenciador. Se o novo sequenciador for o próprio servidor s_k, então o servidor vai reiniciar a ordenação baseando-se nas mensagens já aceitas nos RCDs. Caso não seja s_k o novo sequenciador, s_k apenas aguarda pelas novas propostas.

Com a escolha de um novo sequenciador, o protocolo faz progresso como ocorre na operação normal, isto é, com a ausência de faltas.

5. Provas de Correção

Nesta seção demonstramos que os algoritmos apresentados neste trabalho satisfazem as propriedades especificadas na Seção 3.2. Mais precisamente, o protocolo de difusão

atômica é correto se satisfazer as propriedades definidas como **DA1** a **DA4**, vejamos:

Teorema 1 *O protocolo de difusão atômica especificado pelos Algoritmos 1 e 2 satisfaz as propriedades de* ***Validade*** *e* ***Acordo*** *(DA1 e DA2).*

Prova (esboço): Para esta prova, consideremos que um cliente correto envia uma mensagem ao sequenciador. Se o sequenciador é correto, então o mesmo irá escrever a mensagem no RCD, e a partir dai a mensagem estará disponível à todos os servidores, conforme se pode observar a partir do código das linhas 1 à 11, do Algoritmo 1. Após receber a proposta, cada servidor delibera se a mesma é válida, e quando a maioria concordar com a mesma, cada servidor correto vai armazená-la para, se necessário, difundi-la de maneira confiável, como pode ser visto nas linhas 12 à 15 do Algoritmo 1, e nas linhas 1 à 8 do Algoritmo 2. Caso o sequenciador não seja correto, o cliente irá reenviar sua mensagem para os demais servidores após o tempo $T_{reenviar}$. Com isso, os servidores corretos irão efetuar a mudança de sequenciador, e o novo sequenciador irá retomar a ordenação. E por fim, após receberem as mensagens ordenadas, os clientes corretos irão entregá-las na ordem estabelecida. □

Teorema 2 *O protocolo de difusão atômica especificado pelos Algoritmos 1 e 2 satisfaz a propriedade de* ***Integridade*** *(DA3).*

Prova (esboço): Pelo Algoritmo 1, especificamente nas linhas 2 à 7 é possível observar que o sequenciador efetua a ordenação das mensagens apenas uma vez, portanto, uma mensagem tem um, e somente um, valor de ordenação. Com isso, os clientes corretos entregam as mensagens apenas uma vez, e na ordem estipulada. E como pode ser visto nas linhas 8 à 10 do Algoritmo 1, e na linha 2 do Algoritmo 2, para garantir que uma mensagem só pode ter sido difundida por seu remetente, o vetor de MACs enviado com a mensagem é validado por cada servidor, a fim de evitar que um cliente ou um servidor possa se passar por outro cliente. Estas asserções provam a propriedade de integridade. □

Teorema 3 *O protocolo de difusão atômica especificado pelos Algoritmos 1 e 2 satisfaz a propriedade de* ***Ordem total*** *(DA4).*

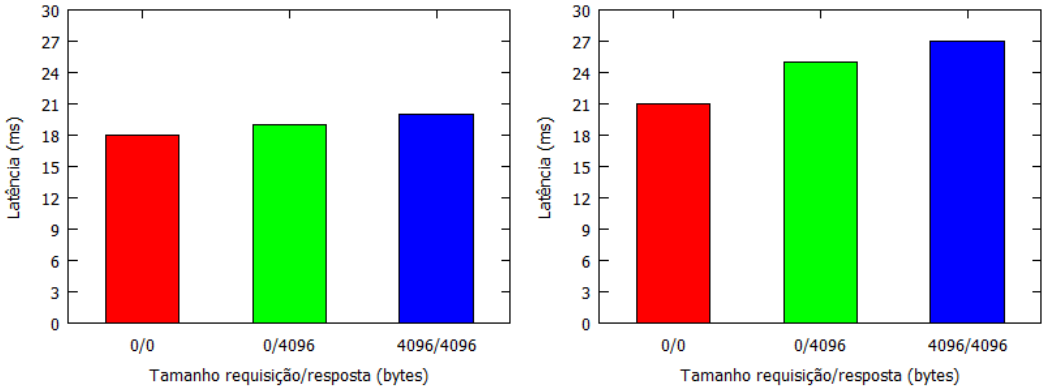
Prova (esboço): Pelo que se pode verificar nas linhas 11 à 15 do Algoritmo 1, e 1 à 8 do Algoritmo 2, respectivamente, se observa que o sequenciador apenas efetua a difusão confiável da mensagem, após ter sido executado o consenso. As mensagens difundidas são aquelas em que a ordenação foi aceita por pelo menos $f+1$ servidores, e a ordem de entrega é determinística. Como o sequenciador difunde a mensagem de maneira confiável, juntamente com sua ordenação e o vetor de MACs para atestar que aquela ordem é aceita pela maioria, consequentemente todos os processo corretos entregarão a mensagem na ordem estipulada, o que prova a propriedade em questão. □

6. Implementação, Avaliação e Resultados

Os algoritmos foram implementados usando a linguagem Java, com o uso do JDK 1.6.0. Os canais de comunicação foram implementados usando *sockets* TCP da API NIO. Os sistemas operacionais usados como hospedeiros das máquinas virtuais foram o "MacOSx Lion 10.7.4" e "Ubuntu 12.04", tendo como o monitor de máquinas virtuais o VirtualBox. Nos convidados (i.e. *guests*) das máquinas virtuais utilizou-se o "Ubuntu 12.04" e o "Debian 6

Stable". Para a avaliação do desempenho, escolhemos a métrica baseada na latência, dado que esta é a métrica largamente empregada na avaliação de sistemas computacionais, por representar de maneira simples, a eficiência do sistema [Jain 1991, Castro and Liskov 2002, Yin et al. 2003, Correia et al. 2006, Favarim et al. 2007].

Os valores foram obtidos através de *micro-benchmarks* com diferentes cargas. A latência foi obtida pela medida do tempo de ida e volta da comunicação (ou *round-trip*), o qual foi extraído pela medida do 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 considerar influências externas. E a fim de avaliar a capacidade do protocolo, executamos as simulações com diferentes tamanhos de mensagens.



(a) Operação normal (b) Operação com faltas.

Figura 3. Desempenho verificado para o DifATo.

E tendo por finalidade a avaliação do desempenho do algoritmo na ausência de faltas, executou-se o protocolo em condições normais, e enviando 10.000 requisições através de um único cliente, e com três condições de carga: 0/0KB, 0/4KB e 4/4KB. 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. E para avaliar o algoritmo também em situações de falha, as réplicas foram configuradas para que, quando assumissem o papel de sequenciadores, enviassem uma entre dez propostas incorretas. As Figuras 3(a) e 3(b) apresentam a latência para cada experimento com as diferentes condições de carga. A latência foi obtida pelo cômputo da média entre o tempo observado após as respostas de todas as requisições enviadas. Como podemos observar, a latência apresenta variações mínimas entre diferentes cargas. Isso se explica pelo fato de que, para aumentar a eficiência do protocolo, o acordo é realizado com base em resumos criptográficos (*hash*) das mensagens, de modo que elas são difundidas apenas em dois passos de comunicação, isto é, no primeiro e no último (Figura 2).

Por fim, para avaliar de eficiência do protocolo proposto de maneira analítica, realizamos um estudo comparativo entre o DifATo e o estado da arte em sistemas de difusão atômica. Estes dados são apresentados na tabela 1. É importante salientar que todos os dados consideram apenas as execuções dos protocolos no caso normal, isto é, na ausência de faltas (mesmo no caso dos clientes). Caso se considere clientes faltosos a troca de mensagens em nosso protocolo aumenta para $n_s{}^2 + n_c$. Pelo dados podemos verificar que os

benefícios do uso do DifATo são visíveis quando comparados os números de passos de comunicação, quantidade de servidores necessário, e o número de mensagens trocadas. Nossa abordagem tem a melhor resiliência prática em termos de quantidade de servidores, além disso, são necessários menos passos para realizar a difusão atômica. Também, ao evitar o envolvimento dos clientes, permitimos que um número finito de clientes seja suscetível a falhas de parada.

Tabela 1. Comparação entre as propriedades de protocolos de difusão atômica.

Protocolos Avaliados	***Propriedades e características verificadas***			
	Resiliência	**Passos de comunicação**	**Mensagens trocadas**	**Tipo de faltas**
Rampart [Reiter 1994]	$3f+1$	6	$6n-6$	Bizantina
Guerraoui e Schiper [Guerraoui and Schiper 2001]	-	5	$3n_c+2n_s-3$	parada
Correia e Veríssimo [Correia et al. 2006]	$3f+1$	-	$18n^2+13n+1+16n^2f+10nf$	Bizantina
Pieri et al. [Pieri et al. 2010]	$3f_c+1+2f_s+1$	5	$2(ns^2+3nc-ns-1)$	Bizantina
DifATo	$2f+1$	4	${n_s}^2-n_s+n_c+1$	Bizantina

7. Conclusão

Ao explorar o uso dos registradores compartilhados distribuídos e da tecnologia de virtualização, foi possível propor uma rede inviolável para implementar um protocolo de suporte à difusão atômica tolerante a faltas Bizantinas. Neste sentido, foi mostrado que é possível implementar um serviço de consenso confiável com apenas $2f+1$ servidores a partir do uso de tecnologias comuns, tal como virtualização e abstração de compartilhamento de dados. A tecnologia de virtualização é amplamente utilizada e entrega o isolamento necessário entre os servidores e o exterior, da mesma forma que o uso dos RCDs torna bastante simples a manutenção das propriedades do protocolo. Como trabalhos futuros, está prevista a criação de um mecanismo semelhante ao que é apresentado em [Veronese et al. 2009], a fim de que seja possível tolerar também, faltas arbitrárias oriundas dos clientes.

Referências

Bessani, A. N., da Silva Fraga, J., and Lung, L. C. (2006). Bts: A byzantine fault-tolerant tuple space. In *Proceedings of the 2006 ACM symposium on Applied computing*, pages 429–433. ACM.

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

Chandra, T. D. and Toueg, S. (1996). Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267.

Correia, M., Neves, N. F., 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. F., and Veríssimo, P. (2006). From consensus to atomic broadcast: Time-free byzantine-resistant protocols without signatures. *The Computer Journal*, 49(1):82–96.

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.

Défago, X., Schiper, A., and Urbán, P. (2004). Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys (CSUR)*, 36(4):372–421.

Ekwall, R., Schiper, A., and Urbán, P. (2004). Token-based atomic broadcast using unreliable failure detectors. In *Reliable Distributed Systems, 2004. Proceedings of the 23rd IEEE International Symposium on*, pages 52–65. IEEE.

Favarim, F., Fraga, J. S., Lung, L. C., Correia, M., and Santos, J. F. (2007). Exploiting tuple spaces to provide fault-tolerant scheduling on computational grids. In *Object and Component-Oriented Real-Time Distributed Computing, 2007. ISORC'07. 10th IEEE International Symposium on*, pages 403–411. IEEE.

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

Guerraoui, R. and Schiper, A. (2001). The generic consensus service. *Software Engineering, IEEE Transactions on*, 27(1):29–41.

Jain, R. (1991). *The art of computer systems performance analysis*, volume 182. John Wiley & Sons Chichester.

Kemme, B., Pedone, F., Alonso, G., Schiper, A., and Wiesmann, M. (2003). Using optimistic atomic broadcast in transaction processing systems. *Knowledge and Data Engineering, IEEE Transactions on*, 15(4):1018–1032.

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

Obelheiro, R., Bessani, A., and Lung, L. (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'05*, pages 99–112. SBC.

Pieri, G., da Silva Fraga, J., and Lung, L. C. (2010). Consensus service to solve agreement problems. In *Parallel and Distributed Systems (ICPADS), 2010 IEEE 16th International Conference on*, pages 267–274. IEEE.

Reiter, M. K. (1994). Secure agreement protocols: Reliable and atomic group multicast in rampart. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, pages 68–80. ACM.

Rodrigues, L., Veríssimo, P., and Casimiro, A. (1993). Using atomic broadcast to implement a posteriori agreement for clock synchronization. In *Proceedings of the 12th Symposium on Reliable Distributed Systems - SRDS'93*, pages 115–124. IEEE.

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

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

Veronese, G. S., Correia, M., Bessani, A. N., and Lung, L. C. (2009). Spin one's wheels? byzantine fault tolerance with a spinning primary. In *Reliable Distributed Systems, 2009. SRDS'09. 28th IEEE International Symposium on*, pages 135–144. 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.