# Replicação Máquina de Estados Dinâmica*

**Eduardo Adilio Pelinson Alchieri**[1], **Alysson Neves Bessani**[2] , **Joni da Silva Fraga**[3]

[1] [1]Departamento de Ciência da Computação
Universidade de Brasília
alchieri@cic.unb.br

[2]Faculdade de Ciências
Universidade de Lisboa
bessani@di.fc.ul.pt

[3]Departamento de Automação e Sistemas
Universidade Federal de Santa Catarina
fraga@das.ufsc.br

***Resumo.*** *A replicação Máquina de Estados é a abordagem mais abrangente e também a mais usada na implementação de sistemas tolerantes a faltas, tanto por parada quanto bizantinas. Esta abordagem consiste em replicar os servidores e coordenar as interações entre os clientes e as réplicas dos servidores, com o intuito de que as várias réplicas apresentem a mesma evolução em seus estados. Apesar da grande maioria das concretizações de replicação Máquina de Estados terem sido desenvolvidas para ambientes estáticos, onde todas configurações do sistema como o conjunto de réplicas que o implementa nunca sofre alterações, muitas aplicações necessitam executar reconfigurações para alterar estes e outros parâmetros do sistema. Neste sentido, este artigo descreve nossos esforços para adicionar suporte a reconfigurações no sistema* BFT-SMART*, que é uma implementação de uma replicação Máquina de Estados tolerante a faltas bizantinas.*

***Abstract.*** *State Machine Replication is an approach widely used to implement fault-tolerant systems. The idea behind this approach is to replicate the servers and to coordinate the interactions among clients and servers replicas, making all of these replicas present the same state evolution (changes). Although most of the State Machine Replication implementations were developed for static environments, where system settings (as the servers set) do not change, many applications need reconfigurations in order to change the system parameters and the servers set. In this sense, this paper describes our efforts to add support to reconfigurations in* BFT-SMART*, which is an State Machine Replication implementation able to tolerate Byzantine failures.*

## 1. Introdução

A replicação Máquina de Estados [Schneider 1990], também chamada de RME, é a abordagem mais abrangente e também a mais usada na implementação de sistemas tolerantes a faltas, tanto por parada [Schneider 1990] quanto bizantinas [Castro and Liskov 2002]. Esta abordagem consiste em replicar os servidores e coordenar as interações entre os clientes e as réplicas dos servidores, com o intuito de que as várias réplicas apresentem a mesma evolução em seus estados.

*Este trabalho recebeu apoio do DPP/UnB através do Edital 10/2012.

O ponto fundamental de uma replicação Máquina de Estados é a necessidade de ordenação das requisições dos clientes para serem executadas pelas réplicas (servidores) que implementam o sistema, a fim de manter o determinismo de réplicas. Para isso, é necessário o emprego de um protocolo de difusão atômica, que geralmente é implementado através de um algoritmo de consenso, onde todas as réplicas entram em acordo sobre a ordem de execução das requisições, as quais são então organizadas em uma sequência que é executada no sistema.

A grande maioria das concretizações existentes para replicação Máquina de Estados (ex.: [Castro and Liskov 2002]), bem como os protocolos propostos para difusão atômica (ex.: [Correia et al. 2006]), consideram que o conjunto de réplicas que implementam o sistema, além dos parâmetros de configuração do mesmo, nunca sofrem alterações. Uma grata exceção é o sistema SMART [Lorch et al. 2006] que implementa uma replicação Máquina de Estados onde apenas o conjunto de servidores pode ser alterado, sendo que os mesmos podem falhar apenas por parada.

A abordagem estática para replicação Máquina de Estados impossibilita, por exemplo, que durante a execução do sistema novas réplicas sejam adicionadas e/ou réplicas antigas sejam removidas e/ou que qualquer outro parâmetro de configuração (como o limite de falhas suportadas) seja alterado em tempo de execução. No entanto, muitos sistemas necessitam executar estas ações quando querem aumentar/diminuir o limite de falhas suportadas ou trocar um servidor antigo com uma configuração obsoleta por um servidor atualizado. Para isso, surge a necessidade de *reconfiguração* do sistema, tornando-o apto para a execução em ambientes dinâmicos.

Um sistema reconfigurável fornece as interfaces necessárias para sua reconfiguração, não se preocupando com aspectos relacionados com a escolha tanto do momento da reconfiguração quanto da nova configuração a ser adotada pelo sistema. De qualquer forma, os algoritmos que implementam estes sistemas devem prever a possibilidade de entradas e saídas de réplicas do mesmo, e quando consideramos uma replicação Máquina de Estados estas ações de reconfiguração devem ser sincronizadas com as execuções dos protocolos de difusão atômica do sistema, a fim de permitir que todas as réplicas executem a mesma sequência de operações (ou armazenam estados que reflitam a mesma sequência de operações), preservando assim a consistência de seus estados.

Recentemente, Lamport *et al.* [Lamport et al. 2010] apresentam uma discussão, numa visão bastante ampla e pouco aprofundada (sem apresentar protocolos e nem mesmo se aprofundar nas questões envolvidas em uma reconfiguração), sobre como reconfigurar uma replicação Máquina de Estados. Nesta abordagem, a reconfiguração da RME é realizada através da própria execução da RME, i.e., a própria RME define a nova configuração do sistema. Apesar de bastante intuitiva, existem vários problemas não triviais que devem ser tratados em uma concretização desta reconfiguração. Também em vista disto, ainda não existe nenhuma implementação de uma RME que tolera comportamento malicioso de servidores e possui capacidade de reconfiguração.

Este artigo apresenta nossos esforços para adicionar suporte a reconfigurações no sistema BFT-SMART, que é uma concretização de uma replicação Máquina de Estados tolerante a faltas bizantinas. Esta característica é importante na medida em que nestes sistemas reconfiguráveis aumenta-se significativamente a possibilidade de processos maliciosos estarem presentes no sistema, devido às computações de entradas e saídas de processos. De acordo com nossos conhecimentos, a inclusão desta funcionalidade no BFT-SMART o torna a primeira implementação de uma RME capaz de tolerar servidores maliciosos e de alterar o conjunto de

réplicas do sistema em tempo de execução.

O restante deste artigo está organizado da seguinte forma. A Seção 2 apresenta o nosso modelo de sistema. Os conceitos envolvendo uma replicação Máquina de Estados e o sistema BFT-SMART são descritos na Seção 3. A Seção 4 discute como o suporte a reconfigurações foi introduzido no BFT-SMART. A Seção 5 apresenta algumas discussões sobre os mecanismos propostos e as conclusões do trabalho são apresentadas na Seção 6.

## 2. Modelo de Sistema

Consideramos um sistema distribuído completamente conectado composto pelo conjunto universo de processos $U$, que é dividido em dois subconjuntos: um conjunto infinito de servidores $\Pi = \{s_1, s_2, ...\}$ e um conjunto infinito de clientes $C = \{c_1, c_2, ...\}$. A chegada dos processos segue o modelo de chegadas infinitas com concorrência desconhecida mas finita [Aguilera 2004]. Desta forma, em cada instante de tempo real $t$ da execução, o número de processos executando alguma ação no sistema é desconhecido mas finito. Contudo, processos podem chegar em qualquer momento (chegadas infinitas) e também passarão a participar das computações executadas no sistema.

Os processos do sistema estão sujeitos a *faltas bizantinas* [Lamport et al. 1982], i.e., processos faltosos podem exibir qualquer comportamento, podendo parar, omitir envio ou entrega de mensagens, ou desviar de suas especificações arbitrariamente e trabalhar em conjunto com o objetivo de corromper o sistema. Um processo que apresenta comportamento de falha é dito falho (ou faltoso), de outra forma é dito correto. Como veremos a seguir, o número máximo de faltas toleradas pelo sistema, denotado por $f$, pode ser reconfigurado durante a execução do mesmo. No entanto, sempre é necessário pelo menos $3f + 1$ réplicas para tolerar até $f$ réplicas faltosas no sistema em um dado momento.

Com relação ao modelo de sincronia, consideramos um sistema parcialmente síncrono [Dwork et al. 1988]. A ideia por trás destes modelos é de que o sistema trabalha de forma assíncrona (não respeitando nenhum limite de tempo) a maior parte do tempo. Porém, durante períodos de estabilidade, o tempo para transmissão de mensagens é limitado. Além disso, as comunicações entre os processos são realizadas através de canais ponto-a-ponto confiáveis e autenticados.

Finalmente, cada processo possui um par distinto de chaves (chave pública e privada) para usar um sistema de criptografia assimétrica. Cada chave privada é conhecida apenas pelo seu próprio dono, por outro lado todos os processos conhecem todas as chaves públicas. Cada processo do sistema (cliente ou servidor) possui um identificador único, representado por este par de chaves obtidas junto a uma autoridade certificadora, sendo inviável a obtenção de identificadores adicionais por processos faltosos com o objetivo de lançar um ataque *Sybil* [Douceur 2002] contra o sistema.

## 3. Replicação Máquina de Estados

A replicação Máquina de Estados [Schneider 1990] é a abordagem mais abrangente e também a mais usada na implementação de sistemas tolerantes a faltas, tanto por parada [Schneider 1990] quanto bizantinas [Castro and Liskov 2002]. Esta abordagem consiste em replicar os servidores e coordenar as interações entre os clientes e as réplicas dos servidores, com o intuito de que as várias réplicas apresentem a mesma evolução em seus estados, i.e., em qualquer ponto da execução do sistema distribuído todas as réplicas devem possuir o mesmo estado.

Para isso, a replicação Máquina de Estados exige que todas as réplicas, *(i)* partindo de

um mesmo estado e *(ii)* executando o mesmo conjunto de requisições na mesma ordem, *(iii)* cheguem ao mesmo estado final, o que define o determinismo de réplicas.

O item *(i)* é facilmente garantido, bastando iniciar todas as réplicas com o mesmo estado (i.e., iniciar todas as variáveis que representam o estado com os mesmos valores nas diversas réplicas). Para prover o item *(ii)*, é necessária a utilização de um protocolo de difusão atômica como mostra a Figura 1. O problema da *difusão atômica* [Hadzilacos and Toueg 1994], também conhecido como difusão com ordem total, consiste em fazer com que todos os processos corretos, membros de um grupo, entreguem todas as mensagens difundidas neste grupo na mesma ordem. A Figura 1 mostra um exemplo onde dois clientes estão concorrentemente acessando quatro servidores. Neste caso, através de um protocolo de difusão atômica, suas requisições são entregues e executadas na mesma ordem pelos servidores, i.e., caso um servidor execute primeiro a requisição $op1$ (requisição do cliente 1) de depois a requisição $op2$ (requisição do cliente 2), então todos os outros servidores também executarão primeiramente $op1$ e depois $op2$, mantendo a ordem de entrega/execução.

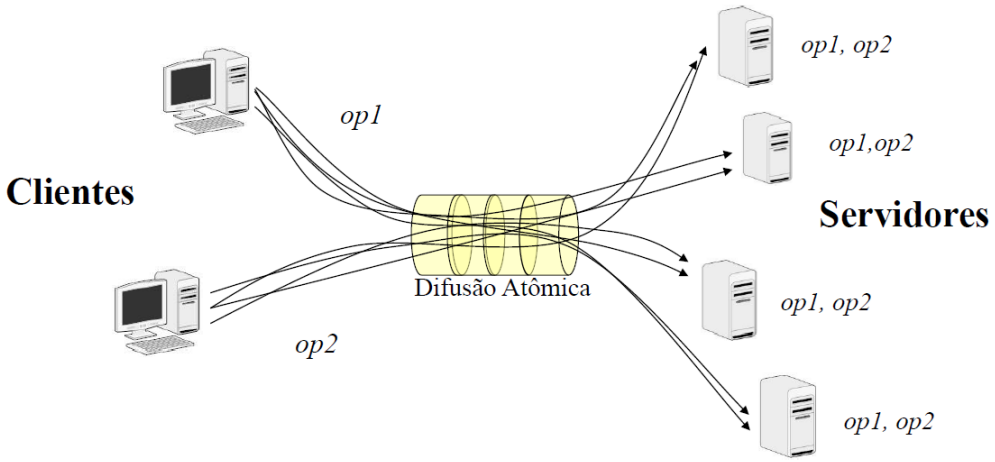


**Figura 1. Replicação Máquina de Estados.**

Finalmente, para alcançar o item *(iii)*, é necessário que as operações executadas pelas réplicas sejam deterministas, i.e., que a execução de uma mesma operação (com os mesmos parâmetros) resulte no mesmo resultado nas diversas réplicas. Além disso, o estado produzido (a mudança no estado) por esta operação deve ser o mesmo nas várias réplicas do sistema.

Um resultado teórico interessante é que a difusão atômica e o consenso são problemas equivalentes em sistemas distribuídos onde os processos então sujeitos tanto a faltas de parada [Chandra and Toueg 1996] quanto a faltas bizantinas [Correia et al. 2006]. O *problema do consenso* [Hadzilacos and Toueg 1994] consiste em fazer com que todos os processos corretos acabem por decidir o mesmo valor, o qual deve ter sido previamente proposto por algum dos processos do sistema. Por exemplo, para implementar difusão atômica através de um protocolo de consenso, basta que os processos utilizem este protocolo para entrarem em acordo (propriedade fundamental do consenso [Hadzilacos and Toueg 1994]) acerca da ordem de entrega das mensagens (requisições).

### 3.1. BFT-SMaRt: Implementação de Replicação Máquina de Estados

O BFT-SMART [Bessani et al. 2011] representa a concretização de uma replicação Máquina de Estados [Schneider 1990] tolerante a faltas bizantinas [Lamport et al. 1982]. Esta biblioteca

de replicação foi desenvolvida na linguagem de programação Java e implementa um protocolo similar aos outros protocolos para tolerância a faltas bizantinas (ex.: [Castro and Liskov 2002]), mas que se preocupa tanto com o desempenho do sistema quanto com a corretude do mesmo nos mais diversos cenários originados pelo comportamento malicioso de réplicas.

O BFT-SMART surgiu da camada de replicação do sistema DEPSPACE [Bessani et al. 2008], o qual representa a implementação de um espaço de tuplas tolerante a faltas bizantinas que utiliza replicação Máquina de Estados para garantir a consistência dos estados das réplicas do sistema. Atualmente, o BFT-SMART conta com protocolos para *checkpoints* e transferência de estados, tornando-se assim uma biblioteca completa para RME, a qual foi desenvolvida seguindo os seguintes princípios:

- Java: a escolha desta linguagem de programação visa a obtenção de uma implementação que apresenta características de portabilidade, segurança, facilidade de programação e manutenção.
- Modularidade: o BFT-SMART foi projetado de forma modular, apresentando uma notável separação entre os protocolos de consenso (o algoritmo de consenso utilizado é o *Paxos at War* [Zielinski 2004]), de difusão atômica, de *checkpoints* e de transferência de estado.
- Desprovido de otimizações que aumentam a complexidade dos algoritmos: além de adicionar complexidade, a utilização de otimizações "frágeis" torna os protocolos mais susceptíveis a ataques de degradação de performance [Clement et al. 2009]. Desta forma, o BFT-SMART não implementa algumas otimizações como a execução do acordo sobre *hashes* e especulação.

Estes princípios de projeto fazem do BFT-SMART a concretização de uma biblioteca de replicação razoavelmente estável e completa, que pode ser usada tanto em pesquisas quanto no desenvolvimento de protótipos.

### 3.1.1. Arquitetura Básica do BFT-SMART

Esta seção discute os aspectos principais da arquitetura do BFT-SMART, os quais são importantes para entender a forma de como as reconfigurações foram incorporadas neste sistema. Primeiramente, vamos analisar como é a arquitetura de cada réplica do sistema (Figura 2). Os clientes acessam as réplicas através de um conjunto de *threads* (uma *thread* é iniciada para cada cliente), sendo que as requisições de cada cliente são adicionadas em uma fila separada. Sempre que existirem requisições para serem executadas, uma *thread* (*proposer*) iniciará uma instância do consenso para definir uma ordem de entrega da mesma (como veremos adiante, na verdade cada instância do consenso define a ordem de execução de várias requisições). Durante este processo, a réplica se comunica com as outras através de um outro conjunto de *threads*, sendo que cada uma destas *threads* é responsável pela conexão com uma das outras réplicas. Além disso, existe um conjunto de *threads* (também uma para cada réplica) responsável por receber as mensagens enviadas pelas outras réplicas. Todo o processamento necessário para garantir a autenticidade destas mensagens é executado nestas *threads*. A utilização destes conjuntos de *threads* faz com que o BFT-SMART apresente um bom desempenho em CPUs multicore.

Finalmente, quando a ordem de execução de uma requisição é definida, a mesma é adicionada em uma fila para então ser entregue à aplicação (*ServiceReplica*) por uma outra *thread*. Após o processamento da requisição, uma resposta é enviada ao cliente que solicitou tal requisição. O cliente, por sua vez, determina que uma resposta para sua requisição é válida

assim que o mesmo receber pelo mesmo $f + 1$ respostas iguais, garantindo que pelo menos uma réplica correta obteve tal resposta.

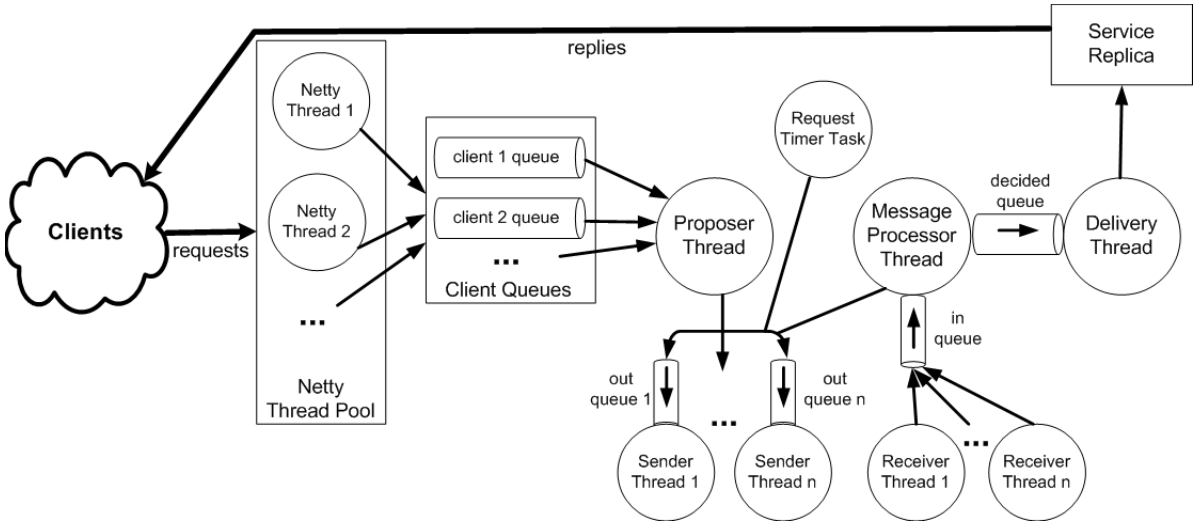


**Figura 2. Arquitetura de uma Réplica do BFT-SMaRt.**

Como já comentado, o protocolo de ordenação das requisições utiliza o algoritmo de consenso *Paxos at war* [Zielinski 2004]. Basicamente, este protocolo funciona da seguinte forma: o valor de decisão da instância $i$ do *Paxos* é a $i$-ésima requisição a ser entregue para a aplicação. Desta forma, a entrega das requisições segue a mesma ordem nas diversas réplicas. Apesar da aparente simplicidade, alguns outros problemas devem ser tratados: *(1)* no protocolo de consenso, a réplica líder pode propor qualquer valor de tal forma que a decisão não seja uma requisição válida e autêntica (requisições forjadas); e *(2)* um líder malicioso pode executar o protocolo de consenso corretamente, mas nunca propor uma ordem para requisições de determinado(s) cliente(s) (negação de serviço para clientes). No BFT-SMaRt, os seguintes mecanismos foram incorporados ao sistema para evitar que réplicas maliciosas sejam capazes de executar qualquer uma destas ações:

- Autenticidade de Requisições: a autenticidade das requisições é garantida por meio de assinaturas digitais, i.e., os clientes devem assinar suas requisições. Desta forma, qualquer réplica é capaz de verificar a autenticidade das requisições e uma proposta para ordenação, a qual contém a requisição a ser ordenada, somente é aceita por uma réplica correta após a autenticidade desta requisição ser verificada.
- Garantia de Ordenação de Requisições: caso uma requisição não seja ordenada dentro de um determinado tempo, o sistema força a troca da réplica líder. Para cada requisição $r$ recebida em determinada réplica $i$, um tempo limite para ordenação é associado à $r$. Caso este tempo se esgotar, $i$ envia $r$ para todas as réplicas e define um novo tempo para sua ordenação. Isto garante que todas as réplicas recebem $r$, pois um cliente malicioso pode ter enviado $r$ apenas para alguma(s) réplica(s), tentando forçar uma troca de líder. Caso este tempo se esgotar novamente, $i$ solicita a troca de líder, que apenas é executada após $f + 1$ réplicas solicitarem esta mudança, impedindo que uma réplica maliciosa force trocas de líder.

A escolha do método de reconfiguração do sistema está diretamente relacionada com a forma de como o protocolo de ordenação das requisições é implementado no mesmo [Lamport et al. 2010]. Neste sentido, a seguir descrevemos duas características fundamentais do protocolo de ordenação do BFT-SMaRt:

- Ordenação em lote: no BFT-SMaRt, cada instância do consenso define a ordem de entrega de um lote de requisições ao invés de apenas uma única requisição. As requisições de um lote devem ser entregues seguindo uma mesma ordem em todos as réplicas (ex.: entregues de acordo com a ordem de seus identificadores). Esta abordagem aumenta o desempenho do sistema, uma vez que várias requisições são entregues através da execução de uma única instância do consenso. No entanto, é possível que requisições de reconfiguração do sistema (que são ordenadas juntamente com as requisições de clientes – Seção 4.1) estejam localizadas no meio de determinado lote, "misturadas" com requisições de clientes, e isto deve ser previsto pelos protocolos de reconfiguração, como veremos na Seção 4.1.
- Eliminação da concorrência na execução de instâncias do consenso: outra abordagem que visa aumentar o desempenho do sistema é a execução em paralelo de várias instâncias do algoritmo de consenso, de modo que várias requisições (ou lotes de requisições) sejam ordenadas simultaneamente. Além desta abordagem trazer mais complexidade aos protocolos, a mesma possibilita que líderes faltosos degradem o desempenho do sistema, uma vez que é necessário executar instâncias do consenso (iniciadas por estes processos maliciosos) mesmo quando as propostas são para definir a ordem de entrega de requisições forjadas. Neste caso, a instância do consenso define a ordem para uma requisição *nop* (*no operation*), o que é necessário para não "travar" a entrega das mensagens [Castro and Liskov 2002]. Em vista disso, o BFT-SMaRt opta por executar uma única instância do consenso por vez e preserva o desempenho do sistema através da ordenação em lotes [Bessani et al. 2008]. Como descrito por Lamport *et al.* [Lamport et al. 2010], esta abordagem também torna o protocolo de reconfiguração menos complexo (Seção 4.1) quando comparado com soluções onde várias instâncias do consenso são executadas em paralelo.

### 3.1.2. Usando o BFT-SMaRt

A forma de utilização do BFT-SMaRt, para programação de uma aplicação tolerante a faltas bizantinas através de replicação Máquina de Estados, é bastante simples. A Figura 3 apresenta a API para clientes e servidores, mostrando a classe que deve ser instanciada pelo clientes para acessar o sistema, bem como a classe que deve ser estendida pelos servidores para implementar o serviço replicado.

Para acessar o serviço replicado, um cliente do BFT-SMaRt apenas deve instanciar uma classe *ServiceProxy* com um arquivo de configuração contendo o endereço (IP e porta) de cada um dos servidores, bem como suas chaves públicas. Então, sempre que o cliente desejar enviar alguma requisição para as réplicas (servidores), o mesmo deve invocar o método *invoke* especificando a requisição (serializada em um *array* de *bytes*) e indicando se tal requisição é apenas de leitura. Requisições de apenas leitura não modificam o estado das réplicas e, deste modo, não precisam ser ordenadas, sendo entregues diretamente para a aplicação.

Por outro lado, para implementar o servidor, cada réplica deve estender a classe *ServiceReplica* e implementar os métodos abstratos que são invocados quando uma requisição deve ser executada (é entregue pelo protocolo de ordenação) ou quando é necessário obter/atualizar o estado da réplica. Os métodos para atualização ou obtenção dos estados das réplicas são utilizados pelos mecanismos de *checkpoints* e transferência de estados, sendo indispensáveis para a recuperação de réplicas faltosas ou atualização de réplicas atrasadas [Bessani et al. 2011].

Note que o método *executeCommand* também fornece o identificador do cliente, um

*timestamp* e um conjunto de *nonces* randômicos, definidos pela réplica líder da instância do consenso que definiu a ordem de execução da requisição correspondente. Como é garantido que todas as réplicas recebem a requisição com o mesmo *timestamp* e conjunto de *nonces*, é possível implementar ações tipicamente não deterministas, como leitura do valor do *clock* ou geração de números randômicos, de forma determinista nas réplicas.

```
//API do Cliente
public class ServiceProxy ... {
  ...
  public byte[] invoke(byte[] command, boolean readOnly);
  ...
}


//API do Servidor
public abstract class ServiceReplica ...{
   ...
   public abstract byte[] executeCommand(int clientId, long timestamp,
                                                 byte[] nonces, byte[] command);
   public abstract byte[] serializeState();
   public abstract byte[] deserializeState(byte[] state);

}
```

**Figura 3. API do BFT-SMaRt para clientes e servidores.**

A Figura 4 apresenta as interações que ocorrem no sistema para a execução de uma requisição que deve ser ordenada, i.e., uma requisição que altera o estado das réplicas. Primeiramente, o cliente envia a requisição assinada para todas as réplicas do sistema, através da classe *ServiceProxy* (método *invoke*). Após receber a requisição, as réplicas executam o protocolo de ordenação para definir uma ordem de entrega para a mesma (passo 2).

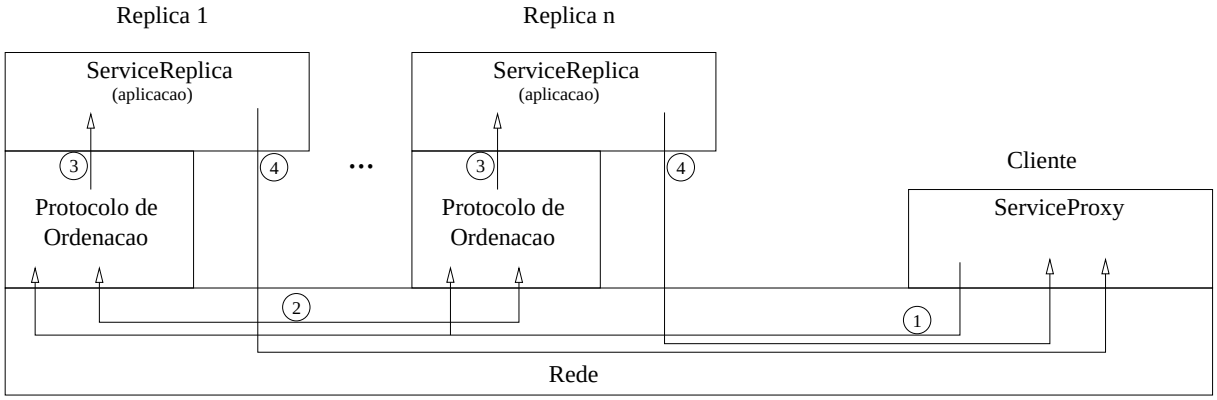


**Figura 4. Interações no BFT-SMaRt.**

Quando as requisições anteriores já tiverem sido entregues em determinada réplica, a mesma executa o método *executeCommand* para entregar esta requisição para a aplicação (*ServiceReplica* - passo 3). Após executar a requisição, cada réplica envia a resposta para o cliente (passo 4), que recebe estas mensagens na classe *ServiceProxy*. Quando pelo menos $f + 1$ respostas iguais são recebidas, o protocolo termina e esta resposta é o resultado da execução do método *invoke*.

## 4. Reconfiguração de uma Replicação Máquina de Estados

Diversas maneiras de reconfigurar o conjunto de réplicas que implementam uma Máquina de Estados são discutidas por Lamport *et al.* [Lamport et al. 2010]. Nestas abordagens, a própria

Máquina de Estados é usada na definição da nova configuração do sistema, fazendo com que todas as réplicas concordem com a nova configuração (visão) de forma semelhante aos protocolos de *group membership* [Chockler et al. 2001].

Seguindo esta abordagem, as reconfigurações do sistema ocorrem entre execuções do protocolo de ordenação (consenso), de modo que a própria RME é usada para definir a nova configuração do sistema. De fato, não faz muito sentido reconfigurar o sistema em meio a uma execução do protocolo de consenso, visto que estes protocolos geralmente terminam rapidamente (em alguns milissegundos – [Castro and Liskov 2002, Bessani et al. 2008]) e a RME em execução já fornece um suporte bastante forte que pode ser utilizado nas reconfigurações. Reconfigurar o sistema em meio a uma execução do consenso implicaria em adicionar muita complexidade neste protocolo, além de atrelar o sistema a um protocolo de consenso específico, i.e., com capacidade de reconfiguração. Além disso, estas limitações seriam introduzidas no sistema sem ganhos práticos, pois a reconfiguração do mesmo exigiria os seguintes passos:

1. O consenso em execução deveria ser paralizado;
2. O sistema deveria ser reconfigurado, onde a nova configuração possivelmente seria escolhida através de um consenso;
3. Então, o consenso anteriormente paralizado poderia ser finalizado.

Como veremos na seção seguinte, a reconfiguração através da própria RME demanda no máximo uma execução do consenso (ou uma execução do protocolo de ordenação da RME) para definir a nova configuração do sistema. Além disso, pode ser necessário que algum cliente execute novamente sua requisição utilizando a visão (configuração) mais atual do sistema, para evitar o acesso a réplicas obsoletas que já não fazem mais parte do sistema.

### 4.1. Reconfigurando o BFT-SMaRt

A reconfiguração do BFT-SMART segue a ideia de utilizar a própria RME para obter a configuração atualizanda, sendo bastante simples. Em termos gerais, o protocolo funciona da seguinte forma:

1. Primeiramente, algum processo envia uma requisição de reconfiguração do sistema, de forma idêntica a um cliente que acessa o sistema normalmente para a execução de alguma operação pela Máquina de Estados.
2. Esta requisição é ordenada junto com as requisições dos clientes, i.e., a mesma é tratada como uma requisição normal pelo protocolo de ordenação, que definirá a ordem de entrega para um lote de requisições que conterá tal requisição de reconfiguração.
3. Quando este lote de requisições for entregue em determinada réplica, a mesma executa todas as operações normais (requisições executadas pela aplicação) para então executar todas as operações de reconfiguração contidas neste lote. Note que um lote pode conter mais de um pedido de reconfiguração e todas as alterações no sistema, definidas pelas requisições deste lote, são processadas de uma só vez após a execução das requisições normais deste lote.
4. Uma nova configuração do sistema é definida e enviada para os clientes, bem como para as réplicas que estão entrando no sistema.
5. As réplicas que estão entrando no sistema atualizam seus estados a partir do estado das réplicas da configuração antiga, as quais fornecem seus estados obtidos até a execução da reconfiguração.
6. Por fim, todas as réplicas da nova configuração iniciam a execução da Máquina de Estados com o estado atual do sistema.

Como no BFT-SMART instâncias de consenso não são executadas em paralelo, não é necessário se preocupar com instâncias utilizando a configuração antiga, uma vez que a instância de consenso seguinte, que definirá a ordem de entrega para o lote de requisições seguinte, já é inicializada utilizando-se a nova configuração do sistema.

Desta forma, o BFT-SMART com capacidade de reconfiguração podem ser entendido como uma sequência de Máquinas de Estados, onde a máquina anterior na sequência escolhe a configuração da máquina seguinte e é finalizada (parada). Então, a máquina seguinte é inicializada com esta configuração a partir do estado final da máquina anterior, e assim por diante.

Existem duas formas de reconfigurações suportadas pelo BFT-SMART: *(1)* no primeiro tipo de reconfiguração a própria réplica (servidor) solicita sua entrada ou saída do sistema; e no segundo tipo *(2)*, mais abrangente, uma terceira parte confiável (TTP - *trusted third party*) tem o poder de reconfigurar o sistema, requisitando entradas e saídas de réplicas e/ou solicitando alterações nos parâmetros de configuração do sistema, como por exemplo o número máximo de faltas toleradas.

### 4.1.1. Visões

Antes de discutir as formas de reconfiguração do BFT-SMART, vamos definir os componentes que formam as visões (configurações) do sistema, uma vez que este conceito é fundamental em sistemas renconfiguráveis, pois sempre que uma reconfiguração ocorre, uma nova visão mais atual é gerada para o mesmo. Esta nova visão reflete os pedidos de alterações que motivaram a execução da reconfiguração.

Sendo assim, cada visão $v$ do sistema contém um identificador único ($v.id$), que nada mais é do que um contador que é incrementado na medida que reconfigurações são executadas. Desta forma, através de seus identificadores, podemos definir facilmente qual de duas visões é mais atual no sistema.

Além disso, cada visão contém os identificadores das réplicas (servidores) que fazem parte de tal visão, bem como o valor dos parâmetros reconfiguráveis no sistema. Atualmente, o único parâmetro reconfigurável no BFT-SMART, além do conjunto de servidores, é o limite $f$ de falhas suportadas pelo sistema. Para cada visão $v$, o número de réplicas presentes nesta visão $v.n$ deve ser $v.n \geq 3v.f + 1$, onde $v.f$ representa o número de falhas de réplicas de $v$ suportadas pelo sistema.

### 4.1.2. Reconfiguração a partir da própria Réplica

Este tipo de reconfiguração é mais restrita, onde uma determinada réplica (servidor) apenas é capaz de solicitar sua própria entrada e/ou saída do sistema. Deste modo, nenhum outro parâmetro do sistema pode ser alterado. A Figura 5 apresenta os métodos adicionados na API dos servidores (classe *ServiceReplica*), os quais permitem a solicitação de suas entradas (*join*) e/ou saídas (*leave*).

Na execução de qualquer um destes métodos, o servidor acessa o sistema como se fosse um cliente e envia a sua solicitação de reconfiguração, que é processada como descrito anteriormente. Após ordenada, esta requisição não é entregue para a aplicação, mas para um módulo de reconfiguração (*ReconfigurationManager*) onde uma nova configuração (visão) é gerada para o sistema. Por fim, uma resposta é enviada ao solicitante para informar a execução de sua

requisição de reconfiguração.

```
//API do Servidor
public abstract class ServiceReplica ...{
   ...
   public void join();
   public void leave();
}
```

**Figura 5. Métodos adicionais para reconfiguração do BFT-SMART.**

A Figura 6 apresenta as interações neste procedimento, onde os passos $3b$ e $4b$ representam a execução de uma requisição de reconfiguração, enquanto os passos $3a$ e $4a$ referem-se à execução de uma requisição normal. Através do protocolo de ordenação, estes passos são sincronizados em todas as réplicas presentes em determinada visão, i.e., as requisições são entregues para a aplicação ou para o módulo de reconfiguração de forma ordenada, seguindo a mesma sequência em todas as réplicas presentes em determinada visão.

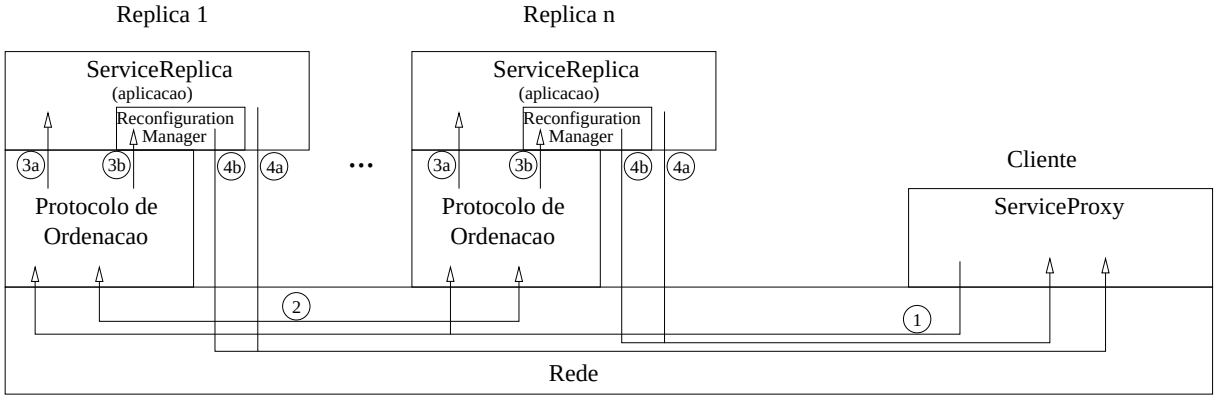


**Figura 6. Interações no BFT-SMART reconfigurável.**

No caso de uma solicitação para entrada no sistema, a requisição contém o identificador e o endereço (IP e porta) do servidor solicitante, de modo que todos os servidores do sistema passam a conhecer este novo servidor. Já a resposta obtida desta solicitação contém a visão na qual o servidor entrou no sistema (nova visão gerada pelo *ReconfigurationManager*) e todos os demais dados necessários para a inicialização da réplica (estado e parâmetros de configuração do BFT-SMART). Além disso, as conexões são atualizadas em todos os servidores do sistema, de acordo com a nova visão do mesmo. Por fim, todos os servidores da nova visão passam a participar do sistema. Todo este processamento é completamente transparente no BFT-SMART, sendo que apenas deve-se executar o método *join* (Figura 5) para que determinada réplica entre no sistema. A computação de um *leave* é mais simples, pois não é necessário atualizar o estado de uma réplica que está deixando o sistema.

### 4.1.3. Reconfiguração a partir da TTP

Este tipo de reconfiguração é mais abrangente, onde uma terceira parte confiável (TTP - *trusted third party*) é capaz de modificar tanto o grupo de réplicas que implementam a Máquina de Estados quanto as configurações do sistema, como por exemplo o limite $f$ de faltas suportadas. A Figura 7 apresenta a API da TTP, onde podemos verificar quais são os métodos que devem ser usados para reconfigurações do sistema.

A TTP deve ser criada da mesma forma de um cliente normal (Seção 3.1.2). Posteriormente, para adicionar ou remover servidores, devemos utilizar os métodos *addServer* e *removeServer*, respectivamente. Para cada novo servidor, é necessário informar seu identificador e seu endereço (IP e porta). Também é possível modificar o limite $f$ de faltas suportadas pelo sistema através do método *setF*, onde esta alteração apenas é executada caso o número de réplicas na nova visão $v$ continue obedecendo ao limite $v.n \geq 3v.f + 1$.

```
//API da TTP
public class TTP{
   ...
   public void addServer(int id, String ip, int port);
   public void removeServer(int id);
   public void setF(int f);
   public void executeUpdates();
   public void close();
}
```

**Figura 7. API da TTP.**

Com o intuito de evitar que para cada alteração desejada seja necessário que o sistema execute uma nova reconfiguração, a TTP armazena todas as alterações solicitadas através destes métodos (*addServer*, *removeServer* e *setF*) e envia apenas uma única requisição de reconfiguração contendo todas estas alterações. Este processamento é executado através do método *executeUpdates*, onde a TTP acessa o sistema como um cliente (assim como descrito na seção anterior) para requisitar a reconfiguração do mesmo.

Quando a resposta é recebida na TTP, a mesma envia uma mensagem para os servidores que estão entrando no sistema, informando-os sobre a visão na qual tais servidores entraram no sistema e sobre os demais dados necessários para a inicialização destas réplicas (estado e parâmetros de configuração do BFT-SMART). Após isso, as réplicas atualizam suas conexões, de acordo com a nova visão do sistema. Por fim, todos os servidores da nova visão passam a participar do sistema. Todo este processamento é completamente transparente no BFT-SMART, sendo que apenas deve-se executar o método *executeUpdates* (Figura 7) para que as atualizações solicitadas sejam refletidas no sistema.

Após a execução do método *executeUpdates*, a TTP fica pronta para ser usada novamente em uma outra reconfiguração, onde novas alterações serão executadas no sistema. No entanto, é possível finalizar a TTP através do método *close* que fecha todas as conexões da TTP com os servidores do sistema. De qualquer forma, caso seja necessário, posteriormente é possível criar uma outra TTP para executar uma nova reconfiguração.

## 5. Discussão

Esta seção apresenta algumas discussões sobre aspectos relacionados com os protocolos de reconfiguração do BFT-SMART. Note que os problemas discutidos e as soluções propostas e adotadas em nosso sistema são válidas para qualquer sistema reconfigurável.

### 5.1. Lidando com Reconfigurações

Para evitar o acesso a dados obsoletos, os clientes do BFT-SMART sempre devem acessar a configuração (visão) mais atual do sistema. Desta forma, um cliente $c$ anexa em suas requisições o identificador da sua visão corrente e os servidores verificam (imediatamente antes da execução e após a ordenação da requisição – passo $3a$ ou $3b$ da Figura 6) se tal cliente está utilizando a visão mais atual do sistema. Se este for o caso, a requisição é executada normalmente. Caso

contrário, os servidores enviam uma resposta para $c$ contendo a nova visão, e então, $c$ envia novamente sua requisição considerando esta nova visão do sistema. Este processamento é completamente transparente no BFT-SMART, sendo apenas necessário invocar as operações normalmente no *ServiceProxy* (Seção 3.1.2).

Note que, considerando $v$ a visão atual do sistema, para uma requisição ser ordenada e entregue, a mesma deve atingir pelo menos um servidor correto de $v$, onde a autenticidade desta requisição é comprovada pela assinatura do cliente. Desta forma, um cliente que utiliza uma visão antiga $w$ apenas terá sua requisição ordenada e obterá uma resposta contendo a visão atualizada $v$, caso $w \cap v$ contenha pelo menos um servidor correto. Para relaxar esta necessidade, as visões do sistema deveriam ser armazenadas também em algum lugar padrão, a partir do qual os clientes poderiam obtê-las, como discutido na seção seguinte.

### 5.2. Armazenamento das Visões

Atualmente, as visões do BFT-SMART são armazenadas apenas pelos próprios servidores que implementam o sistema. Desta forma, os clientes apenas atualizam suas visões nos casos onde suas requisições chegam em algum servidor correto da visão atual do sistema, como discutido na seção anterior.

No entanto, é factível que clientes permaneçam um longo período de tempo sem acessar o sistema, de forma que sua visão não contenha nenhum servidor da visão atual. Nestes casos, tais clientes não conseguem atualizar suas visões. Este mesmo problema ocorre com clientes que desejam acessar o sistema somente após um conjunto de reconfigurações levar o sistema para uma visão que não contém nenhum servidor da visão inicial. Note que este problema afeta apenas os clientes, uma vez que os servidores são sempre informados sobre as novas visões.

Para resolver este problema, é necessário que as visões sejam armazenadas em algum lugar padrão, a partir do qual qualquer cliente possa obtê-las. Neste sentido, é necessário que os servidores emitam certificados (conjunto de assinaturas) que atestem a autenticidade de uma visão. Para aumentar o grau de tolerância a faltas, as visões podem ser armazenadas em vários lugares ou em algum serviço tolerante a faltas bizantinas. Nesta abordagem, sempre que um cliente não tenha sua requisição atendida dentro de um determinado tempo, o mesmo deve verificar se está utilizando a visão atual do sistema.

## 6. Conclusões

Este artigo apresentou os nossos esforços na concretização de uma replicação Máquina de Estados dinâmica, onde o conjunto de réplicas e o limite $f$ de faltas suportadas pelo sistema podem sofrer reconfigurações em tempo de execução. Além disso, os mecanismos propostos para reconfiguração também fornecerão suporte para que outros parâmetros de configuração do sistema possam ser alterados durante a execução do mesmo.

Seguimos uma das abordagens para reconfiguração de uma replicação Máquina de Estados apresentada por Lamport *et al.* [Lamport et al. 2010], onde adicionamos mecanismos que implementam esta abordagem no BFT-SMART. Além da reconfiguração propriamente dita, vários outros aspectos que fogem do escopo deste artigo foram considerados em nossa implementação, como a abertura e o término de conexões de forma segura (evitando que múltiplas conexões sejam estabelecidas para determinada réplica), refletindo a visão atual do sistema. Todos os códigos gerados estão disponíveis na página do BFT-SMART [Bessani et al. 2011].

## Referências

Aguilera, M. (2004). A pleasant stroll through the land of infinitely many creatures. *SIGACT News*, 35(2):36–59.

Bessani, A. N., Alchieri, E. A. P., Correia, M., and da Silva Fraga, J. (2008). Depspace: a byzantine fault-tolerant coordination service. *SIGOPS Oper. Syst. Rev.*, 42(4):163–176.

Bessani, A. N., Alchieri, E. A. P., and Souza, P. (2011). Bft-smart: High-performance byzantine-fault-tolerant state machine replication. http://code.google.com/p/bft-smart/.

Castro, M. and Liskov, B. (2002). Practical Byzantine fault-tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 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.

Chockler, G. V., Keidar, I., and Vitenberg, R. (2001). Group communication specifications: A comprehesive study. *ACM Computing Surveys*, 33(4):427–469.

Clement, A., Wong, E., Alvisi, L., Dahlin, M., and Marchetti, M. (2009). Making byzantine fault tolerant systems tolerate byzantine faults. In *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, pages 153–168. USENIX Association.

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).

Douceur, J. (2002). The sybil attack. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems*.

Dwork, C., Lynch, N. A., and Stockmeyer, L. (1988). Consensus in the presence of partial synchrony. *Journal of ACM*, 35(2):288–322.

Hadzilacos, V. and Toueg, S. (1994). A modular approach to the specification and implementation of fault-tolerant broadcasts. Technical report, Department of Computer Science, Cornell University, New York - USA.

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

Lamport, L., Malkhi, D., and Zhou, L. (2010). Reconfiguring a state machine. *SIGACT News*, 41:63–73.

Lorch, J. R., Adya, A., Bolosky, W. J., Chaiken, R., Douceur, J. R., and Howell, J. (2006). The smart way to migrate replicated stateful services. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, pages 103–115, New York, NY, USA. ACM.

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

Zielinski, P. (2004). Paxos at War. Technical Report UCAM-CL-TR-593, University of Cambridge Computer Laboratory, Cambridge, UK.