Implementando Diversidade em Replicação Máquina de Estados

Caio Yuri Silva Costa[1] **and Eduardo Adilio Pelinson Alchieri**[1]

[1]Departamento de Ciência da Computação
Universidade de Brasília

caio.costa@outlook.com,alchieri@unb.br

Abstract. *The security properties of a system could be impaired by an opponent that exploits its vulnerabilities. An alternative to mitigate this risk is the implementation of intrusion-tolerant systems. State Machine Replication (SMR) is widely used in these implementations. However, the proposed solutions do not allow replica implementation diversity, consequently, the same attack could compromise all the system. In this way, this work proposes an architecture to allow diversity in SMR implementations and shows how this architecture was integrated in the* BFT-SMaRt. *A set of experiments shows the practical behavior of the proposed solutions.*

Resumo. *Vulnerabilidades podem comprometer as propriedades de segurança de um sistema quando adequadamente exploradas por um atacante. Uma alternativa para mitigar este risco é a implementação de sistemas tolerantes a intrusões. Uma abordagem muito utilizada para estas implementações é a replicação Máquina de Estados (RME). Porém, as soluções existentes não suportam diversidade na implementação das réplicas, de forma que um mesmo ataque pode comprometer todo o sistema. Neste sentido, este trabalho propõe uma arquitetura para fornecer suporte à diversidade de implementação em RMEs e mostra como a mesma foi integrada no* BFT-SMaRt. *Um conjunto de experimentos mostra o comportamento prático das soluções propostas.*

1. Introdução

Vulnerabilidades podem comprometer as propriedades de segurança de um sistema caso sejam adequadamente exploradas por um atacante. Em vista disso, os sistemas devem ser construídos de forma a tolerarem intrusões [Fraga and Powell 1985, Veríssimo et al. 2003]: o sistema deve permanecer funcionando corretamente mesmo que uma parte do mesmo seja invadida e controlada por um atacante. Funcionar corretamente significa manter suas propriedades, como disponibilidade, integridade e confidencialidade [Avizienis et al. 2004]. Outro fator que pode comprometer estes sistemas são *bugs* de *software* ou *hardware*, desta forma é necessário que estes sistemas também funcionem corretamente caso alguma parte de mesmo esteja produzindo dados incorretos.

Uma abordagem amplamente utilizada na implementação de sistemas tolerantes a falhas, tanto por *crash* [Schneider 1990] quanto Bizantinas [Castro and Liskov 2002] é a Replicação Máquina de Estados (RME) [Schneider 1990]: esta abordagem consiste em replicar os servidores (permitindo que alguns deles falhem, i.e., sejam invadidos e controlados por um atacante ou apresentem algum *bug* de *software* ou *hardware*) 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.

Embora existam muitas propostas de sistemas e protocolos para RME (ex.: [Amir et al. 2011, Castro and Liskov 2002, Clement et al. 2009b, Clement et al. 2009a, Guerraoui et al. 2010, Kotla et al. 2009, Veronese et al. 2013, Bessani et al. 2014], para citar apenas alguns), nenhum deles fornece suporte para implementação de diversidade nas réplicas [Obelheiro et al. 2005, Bessani et al. 2009, Garcia et al. 2011, Platania et al. 2014, Garcia et al. 2014], i.e., implementar diferentes réplicas em diferentes linguagens de programação aumentando o grau de independência de falhas (duas ou mais réplicas não falham pelo mesmo motivo). Sendo assim, nestes sistemas existentes, um atacante pode utilizar o mesmo ataque para comprometer todas as réplicas do sistema ou ainda, como o mesmo *software* está executando nas diversas réplicas, um mesmo *bug* pode comprometer toda a aplicação.

Com o objetivo de preencher esta lacuna, este trabalho apresenta nossos esforços para adicionar suporte a diversidade na implementação das réplicas (e clientes) no BFT-SMaRt [Bessani et al. 2014], que é uma biblioteca escrita em Java para implementação de uma RME tolerante a falhas Bizantinas. Nosso objetivo não é implementar esta biblioteca em diferentes linguagens, mas sim fornecer interfaces para que clientes e réplicas possam ser desenvolvidas em diferentes linguagens. A implementação de uma biblioteca completa para RME demanda um grande esforço financeiro e muito tempo. Por exemplo, demorou mais de 5 anos para tornar o Bft-SMaRt um sistema robusto [Bessani et al. 2014]. Este trabalho busca propor uma alternativa mais viável.

A arquitetura proposta utiliza o conceito de estado abstrato independente da linguagem [Castro et al. 2003] e apresenta pelo menos dois grandes desafios: *(1)* internamente em um processo – possibilitar a comunicação entre diferentes linguagens (ex.: uma réplica escrita em C precisa executar métodos do BFT-SMaRt escritos em Java e vice-versa); e *(2)* entre processos – possibilitar a troca de informações entre réplicas (ou entre réplicas e clientes) implementadas em linguagens diferentes (ex.: a representação de um vetor em C é diferente da representação em Java).

Resumidamente, as principais contribuições deste trabalho são: *(i)* proposta de uma arquitetura para suportar diversidade na implementação de réplicas (e clientes) de uma RME e descrição de como a mesma foi integrada no Bft-SMaRt, *(ii)* discussão de como o ambiente de execução deve ser configurado para aumentar a segurança das aplicações, e *(iii)* apresentação e análise de uma série de experimentos com as implementações realizadas, trazendo uma melhor compreensão a respeito do funcionamento de diversidade em uma RME.

O restante deste texto está organizado da seguinte forma. A Seção 2 discute mais detalhadamente o conceito de diversidade, enquanto que a Seção 3 apresenta a biblioteca para RME (Bft-SMaRt) utilizada neste trabalho. A Seção 4 discute nossa proposta para uso de diversidade no Bft-SMaRt. Vários experimentos são discutidos na Seção 5. Finalmente, a Seção 6 apresenta as conclusões do trabalho bem como os trabalhos futuros.

2. Diversidade

A abordagem de usar diferentes implementações de um sistema (ou parte dele) para tolerar falhas de *software* foi proposta ainda na década de 70 [Randell 1975,

Avizienis and Chen 1977]. A ideia central destas propostas é que implementações diferentes não apresentem as mesmas falhas de *software*. Recentemente, vários trabalhos abordaram (apenas de forma teórica) o uso de diversidade para prover independência de falhas e assim tolerar intrusões [Obelheiro et al. 2005, Bessani et al. 2009, Garcia et al. 2011, Platania et al. 2014, Garcia et al. 2014].

Existem vários pontos de um sistema (ou parte dele) que podem ser diversificados, formando os seguintes eixos de diversidade [Obelheiro et al. 2005]:

1. Implementação: Consiste na implementação de diferentes versões de um *software*.
2. Administração: Consiste na distribuição dos componentes de um sistema em diferentes domínios administrativos.
3. Localização: Consiste em espalhar os vários componentes físicos de um sistema entre diferentes sítios de instalação.
4. COTS (*Commercial Off The Shelf*): Consiste em utilizar vários COTS diferentes que implementam algumas funcionalidades, como compiladores, bibliotecas, etc.
5. Sistema Operacional: Consiste na distribuição da aplicação entre diferentes sistemas operacionas.
6. Métodos: Consiste na utilização de métodos distintos para garantir algum atributo de segurança, como por exemplo cifrar sucessivamente um dado com mais de um algoritmo criptográfico.
7. *Hardware*: Consiste na distribuição da aplicação entre diferentes *hardwares*.

Este trabalho propõe uma arquitetura que possibilita o emprego de diferentes linguagens de programação na implementação de diferentes réplicas de uma RME. Desta forma, a arquitetura proposta fornece suporte principalmente para diversidade de implementação em RMEs, embora seja possível explorar outros eixos de diversidade, como por exemplo o sistema operacional e o *hardware* (Seção 5.2).

3. BFT-SMART: Implementação de Replicação Máquina de Estados

Em uma replicação Máquina de Estados [Schneider 1990], as réplicas devem apresentar a mesma evolução em seus estados: *(i)* partindo de um mesmo estado e *(ii)* executando o mesmo conjunto de requisições na mesma ordem, *(iii)* todas as réplicas devem chegar ao mesmo estado final, definindo o determinismo de réplicas. Para prover *(i)*, basta 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). Garantir o item *(ii)* envolve a utilização de um protocolo de difusão atômica. O problema da *difusão atômica* [Hadzilacos and Toueg 1994] (ou difusão com ordem total) consiste em fazer com que todos os processos corretos de um grupo entreguem todas as mensagens difundidas neste grupo na mesma ordem. Já para prover *(iii)* é necessário que as operações executadas pelas réplicas sejam determinísticas (i.e., a execução de uma mesma operação, com os mesmos parâmetros, deve produzir o mesmo resultado nas diversas réplicas).

O BFT-SMART [Bessani et al. 2014] é uma biblioteca para implementação de aplicações através de replicação Máquina de Estados [Schneider 1990] que tolera falhas Bizantinas em algumas das réplicas (adicionalmente, o sistema pode ser configurado para tolerar apenas *crashes*). Esta biblioteca *open-source* de replicação foi desenvolvida em Java e implementa um protocolo para RME similar aos outros protocolos para tolerância

a falhas Bizantinas (ex.: [Castro and Liskov 2002]). Além disso, são fornecidos protocolos para reconfiguração e para gerenciamento de estados (*checkpoints*, atualização e transferência de estados).

O BFT-SMART assume um modelo de sistema usual para replicação Máquina de Estados [Castro and Liskov 2002, Bessani et al. 2014]: $n \geq 3f + 1$ servidores para tolerar f falhas Bizantinas; um número ilimitado de clientes que podem falhar; e um sistema parcialmente síncrono para garantir terminação. Estes parâmetros (n e f) podem ser alterados durante a execução através de reconfigurações [Alchieri et al. 2013]. Para comunicação, o sistema ainda necessita de canais ponto-a-ponto autenticados e confiáveis, que são implementados usando MACs (*message authentication codes*) sobre o TCP/IP. As chaves simétricas para a comunicação entre as réplicas são geradas através do protocolo *Signed Diffie-Helman* usando um par de chaves RSA para cada réplica. Já as chaves para a comunicação entre clientes e réplicas são geradas com base nos identificadores dos *endpoints* (cliente e réplica), i.e., não é necessário um par de chaves RSA para cada cliente. Adicionalmente, pode-se configurar os clientes para assinar suas requisições, garantindo-se autenticação das mesmas (neste caso, o par de chaves RSA é necessário).

Resumidamente[1], cada réplica do BFT-SMART realiza as seguintes tarefas:

Recebimento de Requisições. Os clientes enviam suas requisições para as réplicas, que as armazenam em filas diferentes para cada cliente. A autenticidade das requisições é garantida por meio de assinaturas digitais, i.e., os clientes assinam suas requisições (caso configurado). Desta forma, qualquer réplica consegue verificar a autenticidade das requisições e uma proposta para ordenação, que contém a requisição a ser ordenada, somente é aceita por uma réplica correta após a autenticidade desta requisição ser verificada.

Ordenamento de Requisições. Sempre que existirem requisições para serem executadas, um protocolo de difusão atômica é executado onde uma instância de um protocolo de consenso é inicializada por uma réplica (chamada de líder) para definir uma ordem de entrega para um lote de requisições. Caso uma requisição não seja ordenada dentro de um determinado tempo, o sistema troca a réplica líder. Um tempo limite para ordenação é associado a cada requisição r recebida em cada réplica i. 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 enviar 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 troca, impedindo que uma réplica maliciosa force trocas de líder.

Execução de Requisições. Quando a ordem de execução de um lote de requisições é definida, este lote é adicionado em uma fila para então ser entregue à aplicação. Após o processamento da cada 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.

[1]Uma descrição mais detalhada do BFT-SMART pode ser encontrada em [Bessani et al. 2014].

3.1. Implementando Aplicações com o BFT-SMaRt

A forma de utilização do BFT-SMaRt, para programação de uma aplicação tolerante a falhas através de RME, é bastante simples. O Algoritmo 1 apresenta a API básica para clientes e servidores, mostrando a classe que deve ser instanciada pelos clientes para acessar o sistema, bem como a interface que deve ser estendida pelos servidores para implementar o serviço replicado.

Algoritmo 1 API do BFT-SMaRt para clientes e servidores.

```
1  //API do Cliente
2  public class ServiceProxy {
3    public ServiceProxy(int id){
4       ...
5    }
6
7    public byte[] invokeOrdered(byte[] request){
8      ...
9    }
10
11   public byte[] invokeUnordered(byte[] request){
12     ...
13   }
14 }
15
16 //API do Servidor
17 public class MyServer extends Executable {
18
19    public MyServer(int id){
20       new ServiceReplica(id, this, ...);
21    }
22
23    public byte[] executeOrdered(byte[] request, MsgContext ctx){
24       //CÓDIGO DA APLICAÇÃO
25    }
26
27    public byte[] executeUnordered(byte[] request, MsgContext ctx){
28       //CÓDIGO DA APLICAÇÃO
29    }
30 }
```

Para acessar o serviço replicado, um cliente do BFT-SMaRt apenas deve instanciar uma classe *ServiceProxy* fornecendo seu identificador (inteiro) e um arquivo de configuração contendo o endereço (IP e porta) de cada um dos servidores. Após isso, sempre que o cliente desejar enviar alguma requisição para as réplicas (servidores), o mesmo deve invocar o método *invokeOrdered* especificando a requisição (serializada em um *array* de *bytes*). Para requisições que não precisam ordenação (ex.: operações de apenas leitura), o método *invokeUnordered* deve ser utilizado.

Por outro lado, para implementar o servidor, cada réplica deve estender a interface *Executable* e implementar o método abstrato *executeOrdered* que é invocado quando uma requisição deve ser executada. O método *executeUnordered* também deve ser implementado para execução de operações que não precisam ordenação. Além disso, é necessário instanciar uma *ServiceReplica* que representa propriamente a réplica, fornecendo o identificador (inteiro) da réplica que é mapeado para uma porta e endereço IP através de um arquivo de configuração.

Esta é a API básica do BFT-SMaRt, a qual já é suficiente para implementar uma aplicação tolerante a falhas. No entanto, esta API é muito mais rica, apresentando também

métodos (*getSnapshot* e *installSnapshot*) para o gerenciamento do estado das réplicas (caso deseja-se empregar recuperação de réplicas). Uma descrição mais aprofundada sobre o BFT-SMART pode ser encontrada em [Bessani et al. 2014].

4. Implementando Diversidade em Replicação Máquina de Estados

Esta seção descreve a arquitetura proposta para implementação de diversidade em replicação Máquina de Estados e discute como o mesma foi integrada no BFT-SMART. Em nossa arquitetura, dois problemas principais precisam ser tratados: *(1)* como fazer linguagens diferentes acessarem métodos e funções umas das outras; e *(2)* como trocar informações (dados) entre linguagens diferentes, que também podem adotar diferentes representações de dados.

Para resolver o problema *(1)*, nossa arquitetura utiliza interfaces nativas fornecidas pelas linguagens utilizadas para fazer o programa em Java (linguagem de desenvolvimento do BFT-SMART) executar e/ou receber chamadas de programas em C e, a partir do C, com outras linguagens. Já para resolver o problema *(2)*, nossa arquitetura representa os dados através de *Protocol Buffers* [Protocol Buffers 2016], desenvolvido pela Google em 2008, cujo objetivo é serializar dados de forma universal em qualquer linguagem de programação.

Desta forma, a alternativa utilizada em nossa solução consiste em carregar a JVM (*Java Virtual Machine*) dentro do espaço de memória do programa (que pode ter sido escrito em outra linguagem de programação), realizando chamadas diretas ao ambiente Java. Esta abordagem foi possível pelo fato do Java dar suporte a este tipo de desenvolvimento através da JNI (*Java Native Interface*), que é uma implementação de FFI (*Foreign Function Interface*) para a JVM. Uma FFI possibilita a uma linguagem de alto-nível realizar chamadas para ou a partir de uma linguagem de baixo nível, sem utilização de facilidades IPC (*Inter-Process Communication*) ou RPC (*Remote Procedure Call*).

Potenciais problemas com essa abordagem são conflitos entre a JVM e as outras linguagens de programação, já que ambas operam dentro do mesmo espaço de memória. Um exemplo de problema desse tipo foi encontrado durante o desenvolvimento desta solução, na interface com a linguagem Python: caso ocorresse alguma exceção não-tratada no lado Python, a JVM encerrava abruptamente com um *segfault*. A solução, neste caso, foi criar um tratamento genérico de exceções que, ao ser ativado, encerra corretamente a aplicação.

4.1. Visão Geral da Arquitetura Proposta

Grande parte das linguagens de programação possuem uma FFI para interoperabilidade com a linguagem C, visto que os sistemas operacionais geralmente expõem sua API em linguagem C, e se faz necessário a utilização de chamadas de sistema caso um programa queira utilizar qualquer recurso do sistema operacional, como arquivos ou *sockets*. Por outro lado, poucas linguagens oferecem interoperabilidade direta com a linguagem Java, e em geral, as que fornecem já são linguagens desenvolvidas sobre a JVM.

Dado este cenário, foi concebida uma camada intermediária em C que reside entre o BFT-SMART e as demais linguagens, viabilizando a implementação da diversidade. Efetivamente, foi implementada uma interface de linguagem de alto nível para linguagem

de alto nível, através da introdução da camada intermediária de baixo nível. Por simplicidade, esta camada foi desenvolvida na linguagem C++ e suas funções exportadas para C (utilizando a funcionalidade "*extern* C").

Conforme já comentado, ainda existe a necessidade de transferência de dados entre diferentes linguagens de programação. Como cada linguagem possui uma forma distinta de representação dos dados (*e.g.* objetos em Java *versus* structs em C), faz-se necessário o desenvolvimento de um protocolo em comum para troca de informações entre as diferentes linguagens de programação. Para tal propósito foi utilizada a biblioteca *Protocol Buffers* [Protocol Buffers 2016]. Esta biblioteca trabalha com uma linguagem própria de descrição de dados: arquivos `.proto` que são compilados pelo *Protocol Buffers*, gerando código em múltiplas linguagens de programação, com funções/métodos para codificação e decodificação de dados em *arrays* de *bytes*.

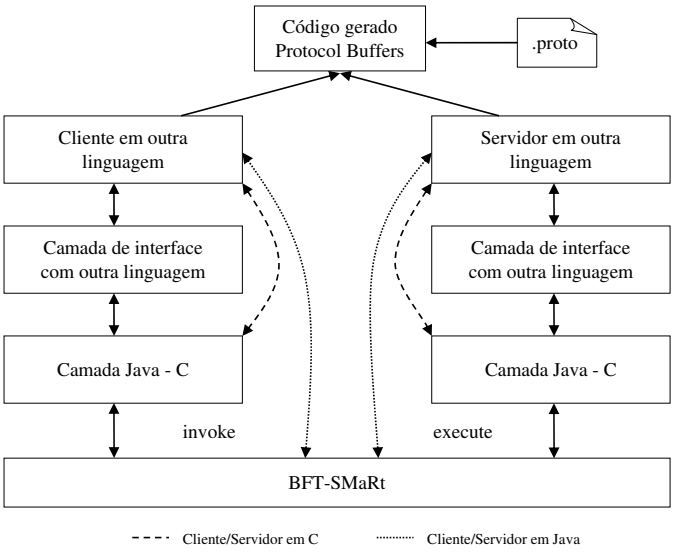

Figura 1. Camadas de interoperabilidade.

A Figura 1 apresenta a arquitetura em camadas da solução proposta. A ideia básica é que clientes escritos em qualquer linguagem utilizem o *Protocol Buffers* para transformar os dados que desejam transmitir (na requisição) em um *array* de *bytes*. Então, é realizada uma comunicação com a linguagem C (camadas intermediárias) que comunica-se tanto com a linguagem na qual foi escrito o cliente quanto com o Java (linguagem em que o BFT-SMART foi escrito). Como as interfaces do BFT-SMART para requisições (e respostas) são definidas através de *arrays* de *bytes* (veja Algoritmo 1), os dados inicialmente transformados em *bytes* pelo cliente através do *Protocol Buffers* são transmitidos até o BFT-SMART que emprega seus protocolos de RME para fazer com que os mesmos sejam entregues para serem executados na mesma ordem pelos servidores. Para isso, é realizado o caminho oposto do descrito anteriormente, i.e., os dados passam do BFT-SMART para as camadas seguintes até chegar na última camada (servidores).

Cada servidor então utiliza o *Protocol Buffers* para reconstituir os dados na sua linguagem de implementação, executa a devida requisição e obtém a resposta. Após isso, para enviar a resposta ao cliente, é realizado o mesmo processamento descrito anteriormente para a requisição. A única diferença é que neste caso o BFT-SMART aguarda por

uma maioria de respostas iguais para então considerar a requisição executada com sucesso e repassar a resposta para as camadas seguintes até chegar ao cliente.

Nesta arquitetura, uma garantia fundamental fornecida pelo *Protocol Buffers*, que possibilita esta integração com o BFT-SMART, é que se os dados são equivalentes então a representação binária dos mesmos é idêntica. Isso possibilita ao BFT-SMART trabalhar somente com os *arrays* de *bytes* (o código em Java do BFT-SMART não possui nenhum conhecimento sobre as estruturas de dados que está transmitindo) e mesmo assim consegue comparar as requisições/respostas e o estado das réplicas para realizar as contagens dos quóruns necessários nos protocolos de RME [Castro and Liskov 2002, Bessani et al. 2014] a fim de manter as propriedades do sistema.

Vale destacar que para clientes ou servidores escritos em Java, as camadas intermediárias não são necessárias. Já para clientes ou servidores escritos em C ou C++, as duas últimas camadas são aglutinadas, i.e., não é necessário utilizar uma FFI para comunicação entre C e estas linguagens.

4.2. Interface com o BFT-SMART

Esta seção descreve em maiores detalhes a interface escrita na linguagem de programação C para comunicação com o BFT-SMART. Em geral, FFIs são compostas por três funcionalidades principais:

a. Utilização de funções escritas em C;
b. Utilização, a partir do C, de funções escritas em outras linguagens; e
c. Conversão de tipos entre as linguagens: entre parâmetros e valores de retorno.

O Algoritmo 2 apresenta a interface em C para comunicação com o BFT-SMART. Tanto clientes quanto servidores, ao iniciar, devem carregar a JVM, opcionalmente configurando um *classpath* customizado, através das funções *setClasspath* e *carregarJvm*.

Algoritmo 2 API do BFT-SMART para clientes e servidores em C.

```
1   void setClasspath(const char* cp);
2   int carregarJvm();
3   void finalizarJvm();
4   int createServiceProxy(int id);
5   int startServiceReplica(int id);
6   int invokeOrdered(BFT_BYTE command[], int tamanho, BFT_BYTE saida[]);
7   int invokeUnordered(BFT_BYTE command[], int tamanho, BFT_BYTE saida[]);
8   void implementExecuteOrdered(int (*impl) (BFT_BYTE [], int, BFT_BYTE **));
9   void implementExecuteUnordered(int (*impl) (BFT_BYTE [], int, BFT_BYTE **));
10  void implementInstallSnapshot(void (*impl) (BFT_BYTE [], int));
11  void implementgetSnapshot(int (*impl) (BFT_BYTE **));
12  void implementReleaseExecuteOrderedBuffer(void (*impl) (BFT_BYTE*));
13  void implementReleaseExecuteUnorderedBuffer(void (*impl) (BFT_BYTE*));
14  void implementReleaseGetSnapshotBuffer(void (*impl) (BFT_BYTE*));
15  void * bftsmartallocate(size_t tamanho);
16  void bftsmartrelease(void * ponteiro) ;
```

Após carregar a JVM, clientes devem invocar a função *createServiceProxy* para criar o *ServiceProxy* utilizado para acessar a RME enquanto que os servidores devem invocar a função *startServiceReplica* para iniciar suas operações (Seção 3.1). A partir deste ponto, os clientes podem realizar requisições através das funções *invokeOrdered* e *invokeUnordered*.

No caso dos servidores, funções escritas em C serão invocadas pelo BFT-SMART. Nesse caso, a assinatura das funções são descritas como métodos em uma classe Java, com a palavra-chave "`native`". Em tempo de execução, cada um desses métodos é associado com um ponteiro para uma função em C. Quando o Java invoca esses métodos, a função correspondente em C é chamada. Sendo assim, servidores devem fornecer uma implementação para as funções *executeOrdered*, *executeUnordered*, *installSnapshot* e *getSnapshot* (estas duas últimas para gerenciamento de estado [Bessani et al. 2014]). Isto é realizado através das funções *implement**, que recebem ponteiros para função.

Além disso, devem ser fornecidas implementações para as funções *implementRelease**. Como não há garantia do tamanho da resposta que vem do servidor, é realizada uma alocação dinâmica de memória, na qual a resposta é armazenada. Após o processamento da resposta pelo ambiente Java, o BFT-SMART chama essas funções para sinalizar que a memória já pode ser liberada.

4.3. Clientes e Servidores com Diversidade

Atualmente, é possível implementar clientes e servidores diversificados em quatro linguagens: Java, C, C++ e Python. A implementação em Java é direta uma vez que o BFT-SMART é escrito em Java, bastando usar o *Protocol Buffers* para codificar os dados e, com isso, possibilitar a troca de dados com réplicas e clientes implementados em outras linguagens. Já para implementações em C e C++, as duas últimas camadas de nossa arquitetura (Figura 1) são aglutinadas, pois não há necessidade de utilização de uma FFI (a comunicação é direta entre a "Camada Java – C" e C ou C++).

Finalmente, a implementação em Python é realizada através da interface apresentada no Algoritmo 2, sendo que foi criada uma interface para realizar as chamadas a esta camada intermediária em C. Deste modo, para implementar clientes e servidores em Python, basta herdar de uma classe abstrata para implementar as funções de forma idiomática, e essa última camada cuida de realizar as traduções devidas para C. Para incluir suporte a outras linguagens de programação, basta utilizar a camada intermediária em C (Algoritmo 2) para comunicação com a nova linguagem de programação.

Vale destacar que os servidores, mesmo implementados em linguagens diferentes, devem apresentar determinismo na execução das operações. A ferramenta DiveInto [Antunes and Neves 2011] possibilita analisar a conformidade entre diferentes implementações de servidores e encontrar inconsistências.

5. Experimentos

Visando analisar o desempenho da arquitetura proposta e da implementação desenvolvida (Seção 5.1), bem como discutir aspectos relacionados com a segurança de aplicações replicadas por meio de uma RME com diversidade (Seção 5.2), alguns experimentos foram realizados no Emulab [White et al. 2002]. O BFT-SMART foi configurado com $n = 4$ servidores para tolerar até 1 falha Bizantina. Cada servidor executou em uma máquina separada, enquanto que 100 clientes executaram em outra máquina. Todos os experimentos realizados tiveram uma fase inicial de *warm-up*.

Aplicação. Uma aplicação de lista, implementada nos servidores como lista encadeada, foi desenvolvida em várias linguagem (Java, C, C++ e Python). Nestes experimentos, a lista foi utilizada para armazenar inteiros e as seguintes operações foram implementadas

para seu acesso: `ADD` – que adiciona um inteiro no final da lista e retorna 1 caso este inteiro ainda não esteja na lista, retorna 0 caso contrário; `REMOVE` – que remove um inteiro passado como parâmetro caso esteja na lista; retorna um código de erro caso contrário; e `GET` – que retorna o elemento da posição passada como parâmetro ou um código de erro caso não exista um elemento na posição indicada (maior que o tamanho da lista).

Em nossos experimentos, a lista foi inicializada com 200k entradas em cada réplica e os valores para as operações `ADD`, `GET` e `REMOVE` foram selecionados aleatóriamente seguindo uma distribuição uniforme. Além disso, todos os clientes executados foram desenvolvidos na linguagem Java, enquanto que os servidores foram diversificados conforme os cenários descritos em cada experimento.

A latência e o *throughput* destas operações foram determinados, visando analisar o desempenho do sistema nos diversos cenários. A latência foi medida em um dos clientes e os valores apresentados representam a média de 1000 execuções, excluindo-se 10% dos valores com maior desvio. Já o *throughput* apresentado é o pico atingido pelos servidores, medido no servidor líder do consenso [Bessani et al. 2014] a cada 5000 requisições.

5.1. Análise do Desempenho: Sem Diversidade no Ambiente de Execução

Primeiramente, executamos alguns experimentos em um ambiente onde todas as máquinas possuiam a mesma configuração, o que possibilitou uma análise mais precisa a respeito do *overhead* introduzido pelo uso de diversidade. Desta forma, o ambiente consistiu de 5 máquinas *d710* (2.4 GHz 64-*bit* Intel Quad Core Xeon E5530, 12GB de RAM e interface de rede gigabit) conectadas a um *switch* de 1Gbps. O ambiente de *software* utilizado foi o sistema operacional Ubuntu 12.04 com *kernel* 3.2.0, JVM Oracle JDK 1.7.0_79 (Java), g++ 4.6.0 (C++), gcc 4.8.3 (C) e Python 2.7.3 (Python).

Resultados e Análises. A Tabela 1 apresenta os valores para o *throughput* e a latência em três cenários distintos: *(1)* apenas usando Java, i.e., o BFT-SMART sem diversidade; *(2)* usando Java e *Protocol Buffers* (Java + PB), que possibilita a avaliação do *overhead* introduzido pelo mecanismo de representação dos dados; e *(3)* usando cada réplica do BFT-SMART em uma linguagem diferente (Java, C, C++ e Python), que possibilita avaliar toda a arquitetura proposta, incluindo o *overhead* tanto do mecanismo de representação dos dados quanto para uma linguagem executar métodos ou funções de outras linguagens.

		Java	Java + PB	Java, C, C++ e Python	Java, C, C++ e C++
`ADD`	Throughput (kop/seg)	11.55	11.32	9.25	10.59
	Latência (ms)	34.92	35.56	100.43	47.46
	Desvio Padrão (ms)	10.10	9.22	46.15	15.48
`GET`	Throughput (kop/seg)	4.45	4.40	0.85	2.08
	Latência (ms)	24.33	24.08	126.63	55.61
	Desvio Padrão (ms)	2.06	2.22	7.73	5.97
`REMOVE`	Throughput (kop/seg)	2.66	2.55	0.48	1.22
	Latência (ms)	47.06	47.91	231.23	90.44
	Desvio Padrão (ms)	5.01	4.19	9.99	7.19

Tabela 1. Experimentos sem diversidade no ambiente de execução.

Podemos perceber que a variação no desempenho introduzida pela utilização de *Protocol Buffers* é praticamente desprezível. Porém, o *throughput* diminui quando o sistema é configurado com uma réplica em cada linguagem, comportamento que fica mais

evidente para as operações `GET` e `REMOVE`. A latência também se comportou de forma semelhante, aumentando quando réplicas em diferentes linguagens foram utilizadas.

Visando analisar os fatores que levaram a esta queda de desempenho, medimos o tempo que cada operação leva para ser executada em cada implementação (execução da operação $executeOrdered$ na linguagem específica para as operação `ADD`, `GET` e `REMOVE`). Como podemos observar na Tabela 2, a implementação em Python apresentou um tempo de resposta muito superior as demais, enquanto C e C++ tiveram desempenhos próximos mas que são praticamente o dobro do que em Java. Vale destacar que estes valores não sofrem influência da arquitetura para diversidade proposta neste trabalho, pois estes valores foram medidos já na linguagem específica (Java, C, C++ e Python).

Como a réplica em Python ficava muito atrasada em relação as demais, a mesma iniciava os protocolos para transferência e atualização de estados [Bessani et al. 2014]. Durante este procedimento, esta réplica solicitava o estado que era transferido a partir das outras réplicas, o que acaba impactando no desempenho do sistema. Para evidenciar este comportamento, na última coluna da Tabela 1 são apresentados os resultados para estes experimentos trocando a réplica em Python por outra réplica em C++. Comparando com a utilização de apenas Java, podemos perceber que neste caso o desempenho melhora, ficando muito próximo para a operação de `ADD` e praticamente a metade para as outras operações, o que é explicado pelo fato de as operações demorarem praticamente o dobro em C++ (Tabela 2). O *throughput* da operação `ADD` fica muito próximo da configuração com apenas Java porque os resultados apresentados referem-se ao pico atingido pelo sistema e, como inicialmente a lista está vazia, este valor é conseguido logo no começo (a queda no desempenho em Python está relacionada com as buscas que ocorrem na lista).

		Java	C	C++	Python
`ADD`	Tempo de execução (ms)	0.33	0.71	0.55	11.45
	Desvio Padrão (ms)	0.09	0.17	0.13	2.45
`GET`	Tempo de execução (ms)	0.20	0.70	0.43	7.68
	Desvio Padrão (ms)	0.10	0.32	0.10	3.36
`REMOVE`	Tempo de execução (ms)	0.51	1.09	0.67	14.03
	Desvio Padrão (ms)	0.06	0.19	0.12	0.51

Tabela 2. Tempo de resposta das linguagens (execução de $executeOrdered$).

Para certificar-se de que a queda de desempenho está relacionada com o processamento da requisição em uma determinada linguagem, analisamos uma aplicação vazia (nada é processado nos servidores, que apenas retornam uma resposta). Este tipo de aplicação é comumente utilizado para avaliar estes sistemas [Bessani et al. 2014] e apenas os tamanhos das requisições/respostas são configurados. A Tabela 3 apresenta os resultados para a configuração com requisições/respostas de tamanho $0/0$. Podemos perceber que neste caso o desempenho é praticamente o mesmo em todos os cenários.

	Java	Java + PB	Java, C, C++ e Python
Throughput (kops/seg)	48.55	48.35	47.17
Latência (ms)	2.24	2.28	2.29
Desvio Padrão (ms)	0.23	0.29	0.36

Tabela 3. Experimento para um aplicação vazia ($0/0$).

5.2. Análise da Segurança: Com Diversidade no Ambiente de Execução

Apesar da aplicação apresentar diversidade de implementação nos experimentos anteriores, a mesma configuração de execução foi utilizada em cada servidor de forma que uma mesma vulnerabilidade pode comprometer toda a aplicação. Por exemplo, uma vulnerabilidade no sistema operacional utilizado (Ubuntu) pode ser explorada em todas as réplicas, visto que todas executam sobre este mesmo sistema operacional. Neste sentido, esta seção apresenta experimentos executados em um ambiente que contempla outros eixos de diversidade, o que aumenta a segurança das aplicações conforme discutido a seguir.

Configuração do Ambiente de Execução e Análise da Segurança. Visando aumentar a segurança através de diversidade, os ambientes de execução das réplicas devem ser diferentes, de forma que seja pouco provável que uma mesma vulnerabilidade esteja presente em mais de um deles. A Tabela 4 apresenta a configuração adotada em cada réplica, que novamente foram conectadas a um *switch* de 1Gbps. A nomenclatura utilizada para o *hardware* é aquela fornecida pelo Emulab [White et al. 2002]. Dentre as configurações disponibilizadas pela plataforma, utilizamos uma configuração diferente para cada servidor. Também utilizamos diferentes distribuições e/ou versões de sistemas operacionais, além de compiladores e/ou ambientes de execução para as linguagens utilizadas. O C++ está presente em todas as réplicas (menos em Java) visto que a camada intermediária foi desenvolvida em C++ e teve sua interface exportada para C (Seção 4).

Eixo de Diversidade		Réplica 1	Réplica 2	Réplica 3	Réplica 4
Aplicação		Java	C++	C	Python
Sistema Operacional		Fedora Core 15 *kernel* 2.6.40	FreeBSD 10.0	CentOS 7.1 *kernel* 3.10.0	Ubuntu 12.04 *kernel* 3.2.0
COTS	JVM (Java)	Oracle JDK 1.7.0_79	OpenJDK 1.8.0_60	Oracle JDK 1.8.0_60	OpenJDK 1.7.0_91
	C	–	–	gcc 4.8.3	–
	C++	–	clang 3.3	g++ 4.8.3	g++ 4.6.3
	Python	–	–	–	Python 2.7.3
Hardware		pc2400w	pc3000	d820	d710

Tabela 4. Configuração das réplicas com diversidade.

Como podemos perceber, um atacante não é capaz de explorar uma mesma vulnerabilidade para comprometer mais de uma réplica. Por exemplo, uma vulnerabilidade no sistema operacional Ubuntu comprometeria apenas a réplica 4, enquanto que uma falha no *hardware* pc3000 afetaria apenas a réplica 2, e assim por diante. Desta forma, a segurança da aplicação é aumentada na medida em que mais vulnerabilidades são necessárias para comprometer o sistema. De fato, é necessário o comprometimento de mais de uma réplica para que o sistema deixe de funcionar corretamente.

Resultados e Análises. A Tabela 5 apresenta os resultados para o sistema com diversidade nos ambientes de execução. Considerando aspectos de desempenho, o comportamento é semelhante aos experimentos anteriores, apresentando uma queda nos cenários com uso de diversidade. No entanto, quando consideramos aspectos de segurança, o sistema fica mais robusto visto que uma combinação de vulnerabilidades (e não apenas uma) é necessária ser explorada por um atacante para comprometer o sistema.

		Java	Java + PB	Java, C, C++ e Python
ADD	Throughput (kop/seg)	4.97	4.62	1.16
	Latência (ms)	27.32	29.69	131.74
	Desvio Padrão (ms)	6.52	8.89	41.70
GET	Throughput (kop/seg)	10.75	10.82	1.54
	Latência (ms)	9.75	9.55	71.02
	Desvio Padrão (ms)	1.10	0.97	8.40
REMOVE	Throughput (kop/seg)	6.61	6.29	1.05
	Latência (ms)	16.18	16.72	106.63
	Desvio Padrão (ms)	1.68	1.176	13.28

Tabela 5. Experimentos com diversidade no ambiente de execução.

6. Conclusões e Trabalhos Futuros

Este artigo apresentou uma nova abordagem para RME que utiliza diversidade para aumentar a segurança das aplicações. De fato, em um sistema diversificado, é muito pouco provável que uma mesma vulnerabilidade possa ser explorada em mais de uma réplica. A arquitetura proposta para suporte à diversidade foi implementada no BFT-SMART e através de uma série de experimentos verificou-se o comportamento na prática de uma RME com diversidade. De um modo geral, o desempenho do sistema como um todo fica condicionado ao pior desempenho dentre as implementações (linguagens) utilizadas. Porém, a segurança do sistema é aumentada, fator primordial para muitas aplicações. As implementações desenvolvidas estão disponíveis no seguinte repositório: `https://github.com/caioycosta/bftsmart-diversity`.

Como trabalhos futuros, pretende-se implementar suporte para outras linguagens de programação, além de explorar outras aplicações a fim de aumentar a compreensão sobre o funcionamento de uma RME com diversidade. Também pretendemos analisar os efeitos da utilização de diversidade nos clientes, embora este não seja o objetivo principal.

Referências

Alchieri, E. A. P., Bessani, A. N., and da Silva Fraga, J. (2013). Replicação máquina de estados dinâmica. In *Anais do XIV Workshop de Teste e Tolerância a Falhas*.

Amir, Y., Coan, B., Kirsch, J., and Lane, J. (2011). Prime: Byzantine replication under attack. *IEEE Transactions on Dependable and Secure Computing*, 8(4):564–577.

Antunes, J. and Neves, N. (2011). DiveInto: Supporting diversity in intrusion-tolerant systems. In *30th IEEE Symposium on Reliable Distributed Systems*.

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

Avizienis, A. and Chen, L. (1977). On the implementation of n-version programming for software fault tolerance during execution. In *International Computer Software and Applications Conference*.

Bessani, A., Daidone, A., Gashi, I., Obelheiro, R., Sousa, P., and Stankovic, V. (2009). Enhancing fault / intrusion tolerance through design and configuration diversity. In *Proceedings of the 3rd Workshop on Recent Advances on Intrusion-Tolerant Systems*.

Bessani, A., Sousa, J., and Alchieri, E. (2014). State machine replication for the masses with BFT-SMaRt. In *Proceedings of the International Conference on Dependable Systems and Networks*.

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

Castro, M., Rodrigues, R., and Liskov, B. (2003). BASE: Using abstraction to improve fault tolerance. *ACM Transactions Computer Systems*, 21(3):236–269.

Clement, A., Kapritsos, M., Lee, S., Wang, Y., Alvisi, L., Dahlin, M., and Riché, T. (2009a). UpRight cluster services. In *Proceedings of the ACM Symposium on Operating Systems Principles*.

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

Fraga, J. and Powell, D. (1985). A fault- and intrusion-tolerant file system. In *Proceedings of the 3rd Int. Conference on Computer Security*, pages 203–218.

Garcia, M., Bessani, A., Gashi, I., Neves, N., and Obelheiro, R. (2011). Os diversity for intrusion tolerance: Myth or reality? In *Proceedings of the IEEE/IFIP 41st International Conference on Dependable Systems&Networks*, DSN '11.

Garcia, M., Bessani, A., Gashi, I., Neves, N., and Obelheiro, R. (2014). Analysis of operating system diversity for intrusion tolerance. *Software: Practice and Experience*, 44(6):735–770.

Guerraoui, R., Knežević, N., Quéma, V., and Vukolić, M. (2010). The next 700 BFT protocols. In *Proceedings of the ACM SIGOPS/EuroSys European Systems Conference*.

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.

Kotla, R., Alvisi, L., Dahlin, M., Clement, A., and Wong, E. (2009). Zyzzyva: Speculative Byzantine fault tolerance. *ACM Transactions on Computer Systems*, 27(4):7:1–7:39.

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*.

Platania, M., Obenshain, D., Tantillo, T., Sharma, R., and Amir, Y. (2014). Towards a practical survivable intrusion tolerant replication system. In *33rd IEEE International Symposium on Reliable Distributed Systems*, pages 242–252.

Protocol Buffers (2016). Protocol buffers developers. Disponível em *https://developers.google.com/protocol-buffers/*. Último acesso em Abril de 2016.

Randell, B. (1975). System structure for software fault tolerance. In *Proceedings of the International Conference on Reliable Software*.

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

Veronese, G., Correia, M., Bessani, A., Lung, L., and Verissimo, P. (2013). Efficient Byzantine fault tolerance. *IEEE Transactions on Computers*, 62(1).

Veríssimo, P., Neves, N. F., and Correia, M. P. (2003). Intrusion-tolerant architectures: Concepts and design. In *Architecting Dependable Systems*, volume 2677 of *Lecture Notes in Computer Science*. Springer-Verlag.

White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., and Joglekar, A. (2002). An Integrated Experimental Environment for Distributed Systems and Networks. In *Proc. of 5th Symp. on Operating Systems Design and Implementations*.