



Universidade de Brasília

Instituto de Ciências Exatas
Departamento de Ciência da Computação

Diversidade em Replicação Máquina de Estados

Caio Yuri da Silva Costa

Dissertação apresentada como requisito parcial para
conclusão do Mestrado em Informática

Orientador

Prof. Dr. Eduardo Adilio Pelinson Alchieri

Brasília
2019

Ficha catalográfica elaborada automaticamente,
com os dados fornecidos pelo(a) autor(a)

CC837d Costa, Caio Yuri da Silva
Diversidade em Replicação Máquina de Estados / Caio Yuri
da Silva Costa; orientador Eduardo Adilio Pelinson
Alchieri. -- Brasília, 2019.
68 p.

Dissertação (Mestrado - Mestrado em Informática) --
Universidade de Brasília, 2019.

1. Replicação Máquina de Estados. 2. tolerância a falhas
bizantinas. 3. diversidade. 4. segurança. I. Alchieri,
Eduardo Adilio Pelinson, orient. II. Título.



Universidade de Brasília

Instituto de Ciências Exatas
Departamento de Ciência da Computação

Diversidade em Replicação Máquina de Estados

Caio Yuri da Silva Costa

Dissertação apresentada como requisito parcial para
conclusão do Mestrado em Informática

Prof. Dr. Eduardo Adilio Pelinson Alchieri (Orientador)
CIC/UnB

Prof. Dr. Diego F. Aranha Prof.a Dr.a Priscila Solis
Aarhus University Universidade de Brasília

Prof. Dr. Bruno Luigi Macchiavello Espinoza
Coordenador do Programa de Pós-graduação em Informática

Brasília, 24 de junho de 2019

Dedicatória

Dedico aos meus pais, meus eternos incentivadores, e aos meus amigos, que tanto torceram pelo meu sucesso na realização deste.

Agradecimentos

Agradeço ao Professor orientador Eduardo Alchieri, pela oportunidade, pela cobrança, e pela paciência durante os empecilhos na execução deste projeto, e à família e amigos, por todo o suporte, sem os quais a execução deste não seria possível.

Resumo

A dependabilidade e as propriedades de segurança de um sistema podem ser comprometidas por uma falha ou por um atacante que explora suas vulnerabilidades, respectivamente. Uma alternativa para mitigar este risco é a implementação de sistemas tolerantes a falhas e tolerantes a intrusões, em que as propriedades do sistema são preservadas mesmo se alguns de seus componentes falharem (e.g., por uma falha de *software* ou uma falha do ambiente de execução) ou se forem comprometidos por um ataque bem-sucedido. A Replicação Máquina de Estados (RME) é amplamente utilizada para implementar esses sistemas. Na RME, servidores são replicados e requisições de clientes são executadas de maneira determinística e na mesma ordem por todas as réplicas de maneira que o comportamento do sistema mantém-se correto mesmo se algumas delas estiverem comprometidas, já que as réplicas corretas mascaram o mal-comportamento das réplicas com falha. Infelizmente, as soluções propostas para RME não consideram diversidade na implementação e todas as réplicas executam o mesmo software. Conseqüentemente, o mesmo ataque ou bug de software pode comprometer todo o sistema. Tentando contornar este problema, este trabalho propõe uma arquitetura para fornecer suporte à diversidade na implementação de serviços seguros utilizando a abordagem RME. Nosso objetivo não é implementar diferentes versões de uma biblioteca RME para diferentes linguagens de programação, o que demandaria muitos recursos. Ao invés disso, a arquitetura proposta usa com base uma biblioteca RME e provê meios de implementar e executar réplicas (aplicações) em diferentes linguagens de programação. Os principais problemas atacados pela arquitetura proposta são dois: (1) comunicação entre diferentes linguagens; e (2) representação de dados. A arquitetura proposta foi integrada à biblioteca RME BFT-SMART e um conjunto de experimentos mostra sua viabilidade prática.

Palavras-chave: Replicação Máquina de Estados, tolerância a falhas bizantinas, diversidade, segurança

Abstract

The dependability and security properties of a system could be impaired by a system failure or by an opponent that exploits its vulnerabilities, respectively. An alternative to mitigate this risk is the implementation of fault- and intrusion-tolerant systems, in which the system properties are ensured even if some of its components fail (e.g., because a software bug or a failure in the runtime environment) or are compromised by a successful attack. State Machine Replication (SMR) is widely used to implement these systems. In SMR, servers are replicated and client requests are deterministically executed in the same order by all replicas in a way that the system behavior remains correct even if some of them are compromised since the correct replicas mask the misbehavior of the faulty ones. Unfortunately, the proposed solutions for SMR do not consider diversity in the implementation and all replicas execute the same software. Consequently, the same attack or software bug could compromise all the system. In order to circumvent this problem, this work proposes an architecture to allow diversity in the implementation of dependable and secure services using the SMR approach. The goal is not to implement different versions of a SMR library for different programming languages, which demands a lot of resources and is very expensive. Instead, the proposed architecture uses an underlying SMR library and provides means to implement and execute service replicas (the application code) in different programming languages. The main problems addressed by the proposed architecture are twofold: (1) communication among different languages; and (2) data representation. The proposed architecture was integrated in the SMR library BFT-SMART and a set of experiments showed its practical feasibility.

Keywords: State Machine Replication, byzantine fault-tolerance, diversity, security

Sumário

1	Introdução	1
1.1	Motivação	1
1.2	Objetivos	2
1.2.1	Geral	2
1.2.2	Específicos	3
1.3	Organização do Texto	3
2	Fundamentação Teórica	4
2.1	Segurança de Funcionamento em Sistemas Distribuídos	4
2.1.1	Sistemas Distribuídos	4
2.1.2	Segurança em Sistemas Distribuídos	5
2.1.3	Tolerância a Falhas em Sistemas Distribuídos	5
2.2	Replicação Máquina de Estados (RME)	7
2.2.1	BFT-SMART	8
2.3	Diversidade	10
2.4	Artefatos usados na implementação de diversidade no BFT-SMART	12
2.4.1	<i>Foreign Function Interface</i> (FFI)	12
2.4.2	Protocol Buffers	15
2.5	Considerações	17
3	Arquitetura para Diversidade em Replicação Máquina de Estados	18
3.1	Princípios de projeto	18
3.2	Modelo de programação	19
3.3	Interface BFT-SMART	20
3.3.1	Camada Java ↔ C	22
3.3.2	Interface C ↔ outras linguagens	25
3.3.3	Cliente/Servidor em outra linguagem	27
3.4	Considerações Finais	30

4 Experimentos	31
4.1 Aplicação	31
4.2 Análise do Desempenho: Sem Diversidade no Ambiente de Execução	32
4.2.1 Resultados e Análises	32
4.3 Análise da Segurança: Com Diversidade no Ambiente de Execução	34
4.3.1 Configuração do Ambiente de Execução e Análise da Segurança. . . .	34
4.3.2 Resultados e Análises.	35
4.4 Análise Complementar do Desempenho	36
4.4.1 Resultados e Análises	36
4.5 Considerações Finais	37
5 Conclusões	38
5.1 Visão Geral do Trabalho	38
5.2 Revisão dos Objetivos e Contribuições	39
5.3 Perspectivas Futuras	39
Referências	41
Apêndice	44
A Demonstração de funcionamento	45
A.1 Algoritmo da lista encadeada	45
A.2 Protocol Buffers	46
A.3 Cliente	47
A.4 Servidor	47
A.5 Execução	48
A.6 Considerações	49

Lista de Figuras

1.1	Diversidade em BFT-SMR e suas réplicas.	2
2.1	Situação de confusão causada por um traidor. Adaptada de Lamport [28] .	6
3.1	Modelo de programação.	20
3.2	Camadas de interoperabilidade.	21
4.1	Experimentos sem diversidade no ambiente de execução.	32
4.2	Experimentos com diversidade no ambiente de execução.	35
4.3	Experimentos com requisições e respostas de diferentes tamanhos.	37

Lista de Tabelas

4.1 Tempo de resposta para cada linguagem (execução do <code>executeOrdered</code>). . .	33
4.2 Experimento para um aplicação vazia (0/0).	34
4.3 Configuração das réplicas com diversidade.	35

Capítulo 1

Introdução

Neste capítulo são apresentadas a motivação e objetivos deste trabalho, além de descrito como o texto é organizado.

1.1 Motivação

Sistemas computacionais confiáveis devem garantir seu correto funcionamento mesmo com a falha de um ou mais de seus componentes. Funcionar corretamente significa manter suas propriedades, como disponibilidade, integridade e confidencialidade [5]. Para tolerar falhas é comum a implementação de uma ou mais soluções CFT (*crash fault-tolerance*). No entanto, tal medida é suficiente apenas para resistir à falha total de um componente—quando o mesmo se torna indisponível, deixando de se comunicar com o resto do sistema, e não oferece proteção contra um componente que passa a responder de forma maliciosa. Para garantir o funcionamento correto do sistema nessas situações, deve-se utilizar uma abordagem BFT (*byzantine fault-tolerance*—tolerante a falhas bizantinas) [18].

Uma abordagem amplamente utilizada na implementação de sistemas tolerantes a falhas, tanto por parada (*crash*) [38] quanto bizantinas (maliciosas) [11] é a Replicação Máquina de Estados (RME) [38]: esta abordagem consiste em replicar os servidores (permitindo que alguns deles falhem, *i.e.*, sejam invadidos e controlados por um adversário ou apresentem alguma falha 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 [3, 10, 11, 14, 15, 24, 27, 43], nenhum deles fornece suporte para implementação de diversidade nas réplicas [9, 21, 22, 32, 34], implementando diferentes réplicas em diferentes linguagens de programação aumentando o grau de independência de falhas (duas ou mais réplicas não falham pelo mesmo motivo [31]). Sendo assim, nestes sistemas existentes, um adversário

pode utilizar o mesmo ataque para comprometer todas as réplicas do sistema ou ainda, como o mesmo *software* está executando nas diversas réplicas, uma mesma falha pode comprometer toda a aplicação.

1.2 Objetivos

1.2.1 Geral

Com o objetivo de preencher a lacuna no suporte a diversidade em sistemas para RME, este trabalho apresenta nossos esforços para adicionar suporte a diversidade na implementação das réplicas no BFT-SMART [10]. O BFT-SMART é uma biblioteca escrita em Java para implementação de uma RME tolerante a falhas bizantinas. Como mostra a Figura 1.1, a ideia é permitir que as réplicas das aplicações sejam escritas em linguagens de programação diversas, de maneira que uma falha ou vulnerabilidade presente na implementação em Python, por exemplo, afete somente a réplica correspondente enquanto as demais réplicas mantêm o sistema operando corretamente. Além disso, enquanto a possibilidade de desenvolver clientes em diferentes linguagens de programação não aumente a segurança, ainda assim essa implementação é interessante pois permite a integração do BFT-SMR com muitos sistemas legados.

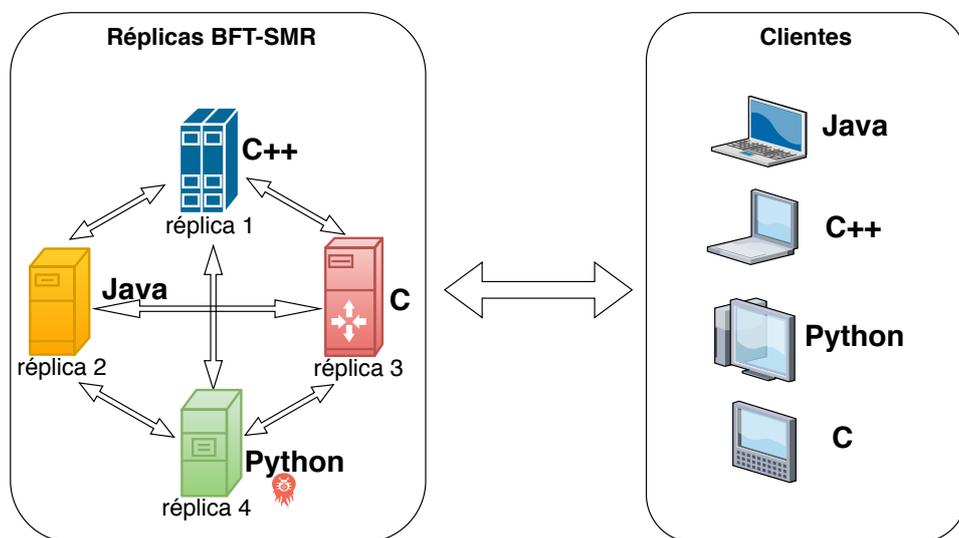


Figura 1.1: Diversidade em BFT-SMR e suas réplicas.

Nosso objetivo não é implementar a diversidade do BFT-SMART em si, reimplementando-o em diferentes linguagens, mas apenas prover os meios para implementar as réplicas com diversidade utilizando esta biblioteca, mantendo a implementação do BFT-SMART

em Java—seria necessário um grande esforço para construir uma implementação de BFT-SMR completa e eficiente. Por exemplo, o BFT-SMART já tem mais de 9 anos de desenvolvimento [10].

A arquitetura proposta utiliza o conceito de estado abstrato independente da linguagem [12] e apresenta pelo menos dois grandes desafios: (1) internamente em uma réplica – possibilitar a comunicação entre diferentes linguagens (*e.g.* uma réplica escrita em C precisa executar métodos do BFT-SMART escritos em Java e vice-versa); e (2) entre réplicas (ou entre réplicas e clientes) – possibilitar a troca de informações entre réplicas implementadas em linguagens diferentes (*e.g.* a representação de um vetor em C é diferente da representação em Java).

1.2.2 Específicos

Os objetivos específicos deste trabalho são as seguintes:

1. Estudar os conceitos envolvendo diversidade e RME.
2. Proposta de uma arquitetura para suportar diversidade na implementação de réplicas de uma RME, que seja extensível (*i. e.*, possibilite a implementação de suporte a novas linguagens de programação) e que apresente baixa perda de desempenho devido à comunicação entre diferentes ambientes de execução, o que também demanda uma forma padrão para representação dos dados.
3. Integração e descrição de como esta arquitetura foi integrada no BFT-SMART.
4. 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.

1.3 Organização do Texto

O texto deste trabalho está dividido em 5 capítulos. Neste primeiro, foram introduzidas a motivação e objetivos. No segundo capítulo, é fornecida uma introdução teórica acerca do tema abordado e das tecnologias utilizadas na implementação dos objetivos. No Capítulo 3 é detalhado o trabalho realizado, a arquitetura da solução proposta e como ela foi integrada ao BFT-SMART. No Capítulo 4 são apresentados os experimentos realizados e seus resultados, e no Capítulo 5 são apresentadas as conclusões deste trabalho. Por fim, o Apêndice A mostra como uma aplicação pode ser desenvolvida utilizando a arquitetura proposta neste trabalho.

Capítulo 2

Fundamentação Teórica

Este capítulo apresenta os conceitos básicos de segurança e falhas em sistemas distribuídos e diversidade, os quais serviram de base para a motivação deste trabalho, o BFT-SMART, plataforma que serviu de base para a implementação prática e por fim as tecnologias utilizadas que possibilitaram a implementação do trabalho.

2.1 Segurança de Funcionamento em Sistemas Distribuídos

Nesta seção serão introduzidos conceitos de falhas e segurança de funcionamento em sistemas distribuídos, e alternativas comuns para contorná-las.

2.1.1 Sistemas Distribuídos

Sistemas distribuídos são aqueles onde entidades independentes interagem, através de protocolos compartilhados, para atingir um objetivo em comum. As características que determinam a natureza de um sistema distribuído são [8]:

1. Entidades múltiplas: Presença de várias entidades participando no sistema, podendo ser usuários ou sub-sistemas.
2. Heterogeneidade: Natureza heterogênea das entidades envolvidas, podendo ser *e.g.* no tipo de usuário ou sistema, dados e recursos.
3. Concorrência: Os diferentes componentes do sistema podem executar simultaneamente.
4. Compartilhamento de recursos: Recursos podem ser utilizados por mais de uma entidade do sistema, necessitando resolver os problemas de sincronização decorrentes.

2.1.2 Segurança em Sistemas Distribuídos

A segurança é, sem dúvida, uma das maiores preocupações em um sistema distribuído. A complexidade deste problema é devido à primeira característica mencionada anteriormente: que o sistema distribuído é composto por múltiplas entidades. Tanto servidores, como clientes e também meios de transmissão precisam ser seguros [8].

Como requisitos para a segurança, temos as seguintes propriedades [5, 8]:

1. Disponibilidade: A informação deve estar disponível a clientes quando solicitada.
2. Confidencialidade: A informação deve ser restrita somente a usuários autorizados, e o acesso por outros deve ser prevenido.
3. Integridade: Durante a transmissão, a informação deve chegar ao seu destino sem ter sido manipulada.

2.1.3 Tolerância a Falhas em Sistemas Distribuídos

Defeitos em *software* e sistemas acontecem quando um sistema se comporta de maneira inesperada, não atendendo à sua especificação. Um erro é a parte do estado do sistema responsável pelo defeito, que por sua vez, é causado por uma falha [29].

Quanto à sua natureza, as falhas podem ser acidentais ou intencionais; quanto à sua causa, podem ser humanas (*e.g.* falhas no código) ou físicas (*e.g.* problemas de *hardware*); quanto a escopo, podem ser internas, como causadas por um estado inconsistente, ou externas, como as causadas por interferência eletromagnética; e a depender do momento que foram introduzidas, podem ser de projeto ou operacionais (por exemplo, as que surgem por uma intrusão) [29].

Dada a diversa gama de fontes de falha em um sistema, é desejável que um sistema seja capaz de funcionar corretamente ainda que uma falha aconteça, *i.e.* é desejável que as falhas não gerem erros. Para tal, o sistema deve ser tolerante a falhas. Um sistema tolerante a falhas possui segurança de funcionamento, a qual é definida através das seguintes propriedades [5]:

1. Disponibilidade: A informação deve estar disponível a usuários quando solicitada.
2. Confiabilidade: O serviço correto das informações deve ser contínuo.
3. Salvaguarda: Ausência de consequências catastróficas para os usuários e o ambiente.
4. Integridade: Durante a transmissão, a informação deve chegar ao destino sem ter sido manipulada.
5. Manutenibilidade: Possibilidade do sistema passar por modificações e reparos.

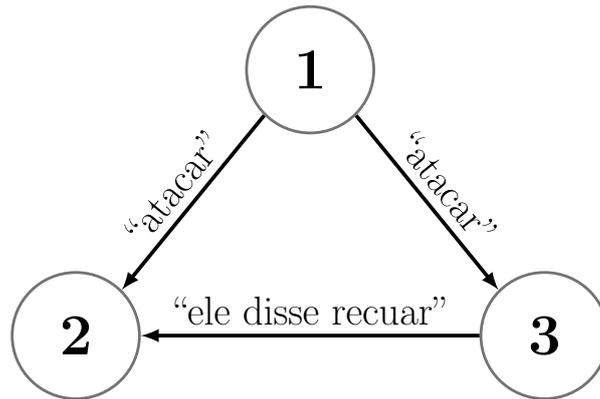


Figura 2.1: Situação de confusão causada por um traidor. Adaptada de Lamport [28]

A implementação mais comum de sistemas tolerantes a falhas é a CFT (*crash fault-tolerance*). Tal classe de falhas consiste na indisponibilidade ou queda de um ou mais nodos de um sistema distribuído. Um tipo de implementação CFT é a duplicação de um componente em duas ou mais máquinas, de modo a garantir que algum problema em alguma delas não interrompa o funcionamento do sistema.

No entanto, uma classe de falhas geralmente não é considerada no projeto de muitos sistemas distribuídos: as falhas bizantinas, para as quais é necessária alguma implementação BFT (*byzantine fault-tolerance*—tolerante a falhas bizantinas). Esta classe de falhas é caracterizada pelo envio de informações conflitantes a diferentes componentes do sistema. Em seu artigo de 1982, Lamport *et al.* [28] descreve esta classe de falhas através de uma abstração com o exército bizantino, de onde vem o nome dessa classe de falhas. A situação é descrita como um determinado número de generais, estacionados com suas tropas ao redor de uma cidade a qual pretendem invadir. Se comunicando somente através de mensageiros, eles devem combinar sua estratégia de ataque. No entanto, alguns dos generais podem ser traidores, que tentarão confundir os outros, conforme exemplificado na Figura 2.1. Nessa situação, devem ser garantidas duas condições:

- Generais leais seguirão o mesmo plano de ação.
- Um pequeno número de traidores não pode levar os não-traidores a seguir uma má estratégia.

Os generais se comunicam através de mensagens orais, que são caracterizadas como:

1. Toda mensagem é entregue corretamente (não há perdas).
2. O receptor sabe quem enviou a mensagem.
3. A ausência de uma mensagem pode ser detectada.

No artigo é provado que para mensagens com essas características, não é possível atingir um consenso. O artigo então apresenta uma solução para o problema, onde os generais trocam informações entre pares e uma votação majoritária faz uma seleção entre todos os dados transmitidos. Essa solução requer que, para tolerar f generais traidores, são necessários $3f + 1$ generais ao todo.

No entanto, como demonstrado no artigo de Kevin Driscoll [18], o conhecimento acerca deste tipo de falha não é difundido na indústria, e é interpretada, geralmente, de forma equivocada, com probabilidade muito baixa de acontecer para demandar um tratamento específico, o qual geralmente é muito custoso.

Há uma tendência do aumento deste tipo de falha dada a evolução do hardware (que está cada vez menos confiável, menos durável, e demandando condições cada vez mais específicas para correto funcionamento, *e.g.* processadores com litografia cada vez menor, sensíveis a variações de temperatura e alimentação elétrica), da disseminação dos sistemas distribuídos e da crescente quantidade de ataques maliciosos contra estes.

O tratamento abstrato dos “generais bizantinos” prejudica a compreensão desse problema pela indústria, pois um engenheiro, que “sabe” que uma CPU não tem vontade própria e portanto não pode mentir, pode desconsiderar o conceito de “generais traidores” se escondendo em seus sistemas [18].

Além dessas ponderações, Driscoll [18] dá vários exemplos reais dessa classe de falhas. Um exemplo simples é a de um circuito digital que fica com um sinal “preso” em uma voltagem intermediária entre 0 e 1, que combinado com o ruído do circuito gera uma resposta diferente para cada leitura.

Uma técnica que fornece tolerância tanto a falhas por parada (*crash*) quanto bizantinas é a Replicação Máquina de Estados (RME). Esta técnica será descrita em maiores detalhes na seção seguinte, uma vez que nosso trabalho concentra-se em adicionar suporte a diversidade na implementação de réplicas em uma RME.

2.2 Replicação Máquina de Estados (RME)

Em uma implementação de Replicação Máquina de Estados [38], 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* [25] (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).

2.2.1 Bft-SMaRt

O BFT-SMART [10] é uma biblioteca para implementação de aplicações através de Replicação Máquina de Estados [38] que tolera falhas bizantinas em algumas das réplicas (adicionalmente, o sistema pode ser configurado para tolerar apenas paradas-*crashes*). Esta biblioteca de replicação, de código-aberto, foi desenvolvida em Java e implementa um protocolo para RME similar aos outros protocolos para tolerância a falhas bizantinas (e.g. [11]). 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 [10, 11]: $n \geq 3f + 1$ 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 [2]. 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. Além disso, pode-se configurar os clientes para assinar suas requisições, garantindo-se autenticação das requisições.

Resumidamente¹, 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. 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.

Ordenamento de Requisições. Sempre que existirem requisições para serem executadas, uma instância do consenso é inicializada por uma réplica (chamada de líder) para

¹Uma descrição mais detalhada do BFT-SMART pode ser encontrada em [10].

definir uma ordem de entrega para um lote de requisições. Durante este processo, a réplica se comunica com as outras através de canais ponto-a-ponto confiáveis e autenticados. 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.

Implementando Aplicações com o Bft-SMaRt

Para programar uma aplicação tolerante a falhas através de RME, a forma de utilização do BFT-SMART 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.

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 vetor de *bytes*). Para requisições que não precisam ordenação (*e.g.* 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

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     public byte[] invokeOrdered(byte[] request){
7         ...
8     }
9     public byte[] invokeUnordered(byte[] request){
10        ...
11    }
12 }
13 //API do Servidor
14 public class MyServer extends Executable {
15     public MyServer(int id){
16         new ServiceReplica(id, this, ...);
17     }
18     public byte[] executeOrdered(byte[] request, MsgContext ctx){
19         //CÓDIGO DA APLICAÇÃO
20     }
21     public byte[] executeUnordered(byte[] request, MsgContext ctx){
22         //CÓDIGO DA APLICAÇÃO
23     }
24 }
```

identificador (inteiro) da réplica que é mapeado para uma porta e endereço IP através de um arquivo de configuração.

Conforme já comentado, esta é a API básica do BFT-SMART que já é suficiente para implementar uma aplicação tolerante a falhas. No entanto, esta API é muito mais rica, apresentando também métodos para o gerenciamento do estado das réplicas (caso deseje-se empregar recuperação de réplicas). Uma descrição mais aprofundada sobre a API do BFT-SMART pode ser encontrada em [10].

2.3 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 [6, 37]. A ideia central destas propostas é que implementações diferentes não apresentem as mesmas falhas de *software*. Recentemente, vários trabalhos abordaram o uso de diversidade como medida de segurança, pois apresenta vantagens como a *independência de falhas*, que garante que diferentes componentes ou réplicas não serão comprometidos simultaneamente, e com isto a *tolerância a intrusões*, que consiste na garantia de que o sistema continue operando corretamente mesmo que uma parte seja comprometida [9, 21, 22, 32, 34].

Um exemplo é a abordagem NVP (*N-version programming*)[13], que consiste na implementação de $N \geq 2$ versões funcionalmente equivalentes de um sistema e que seguem a mesma especificação. As diferentes versões são executadas concorrentemente e um algoritmo de votação é aplicado de modo que, na ocorrência de uma falha em uma das implementações, o resultado correto ainda assim possa ser utilizado e/ou uma resposta negativa seja fornecida, ao invés de um resultado potencialmente incorreto.

A justificativa é que à medida que um *software* aumenta em tamanho e complexidade se torna efetivamente impossível eliminar todos os defeitos antes de colocá-lo em uso. Além disso, a prática comum de duplicar instâncias do *software* para tolerar falhas ocasionais não resolve erros de *software*, que ocorrerão igualmente em todas as réplicas.

Existem vários pontos de um sistema (ou parte dele) que podem ser diversificados, formando os seguintes eixos de diversidade [32]:

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 operacionais.
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 componentes de *hardware*.

Uma alternativa que oferece aumento considerável em segurança a baixo custo é a diversificação do sistema operacional [22, 31]. Isso é possível porque é muito baixo o número de vulnerabilidades que afetam simultaneamente mais de um sistema operacional. Num sistema sem essa diversidade, assim que um invasor compromete uma réplica, ele compromete todo o sistema, pois basta repetir o mesmo ataque para as demais.

Porém, a diversificação de implementação ainda aumenta a segurança contra defeitos deliberados, inseridos intencionalmente, de forma maliciosa, pois provê as seguintes características [26]:

- A multiplicidade de implementações dificulta a inserção da mesma lógica maliciosa em todas as réplicas.
- Garante a completude, já que caso uma versão omita alguma ação, a consequente divergência dos estados é detectada.
- Componentes de *timeout* previnem que a ausência de resposta de um componente cause a parada do sistema (*i. e.*, um ataque de negação de serviço—DoS).

Um problema da diversidade de implementação é o seu alto custo de desenvolvimento, que é multiplicado por N [31]. Uma alternativa, no caso de *softwares* “de prateleira” como bancos de dados e sistemas de arquivos, é utilizar várias implementações de fornecedores diferentes e, através de uma camada de abstração, integrá-los com uma biblioteca BFT [12]. De fato, as soluções para diversidade têm diversas intersecções com as soluções BFT, principalmente:

- Execução em paralelo de réplicas que são funcionalmente equivalentes.
- Algoritmos de consenso para determinação da resposta correta.
- Necessidade de gerenciar a ordem da execução das tarefas para que todas as réplicas estejam sujeitas às mesmas transições de estado.

Portanto, é natural desejar a reutilização das implementações BFT para, também, implementar a diversidade e o NVP em um sistema distribuído.

2.4 Artefatos usados na implementação de diversidade no Bft-SMaRt

2.4.1 *Foreign Function Interface (FFI)*

Frequentemente é necessária a utilização de uma mistura de linguagens de programação: certa funcionalidade pode ser implementada em uma linguagem, enquanto utiliza funções de um sistema legado ou uma biblioteca escrita em outra linguagem. Apesar de muitas vezes essa solução parecer “improvisada”, é na realidade mais eficiente, pois possibilita focar no desenvolvimento de novas funcionalidades ao invés de reimplementar do zero *software* já existente [30].

É possível a utilização de mecanismos como *web services*, troca de mensagens ou *sockets* para fazer essa integração. Porém, além de manter dois recursos separados em execução, a comunicação entre os dois introduz um ponto de falha e de degradação de desempenho.

Para que as linguagens integradas sejam tratadas como um só programa, de tal modo que as duas linguagens de programação compartilhem recursos e memória diretamente, deve ser utilizado o mecanismo de *Foreign Function Interface* (FFI) ou “interface de função estrangeira”. Uma FFI consiste de código que trata detalhes de baixo nível, por exemplo [7]:

- Encapsular objetos de e para código estrangeiro;
- Gerenciar alocação de memória; e
- Tratar diferentes convenções de chamada de função, argumentos implícitos, etc.

As FFIs se diferenciam de outros métodos de comunicação como IPC ou *sockets* pelo fato de que, durante a execução, o código das duas linguagens de programação fará parte de um mesmo processo, compartilhando o espaço de memória—as trocas de informação acontecem através de cópia de memória (*e.g. memcpy*), e portanto com pouco *overhead*.

Para cada linguagem, é necessária uma FFI correspondente. Linguagens de alto nível quase sempre vem acompanhadas de uma FFI para linguagem C, pois como os sistemas operacionais mais comuns possuem API em C, essa FFI é necessária para que a linguagem utilize os serviços do sistema operacional.

Java Native Interface (JNI)

A linguagem Java oferece duas maneiras principais de interação com código escrito em C: o JNI (Java Native Interface) [40] e o JNA (Java Native Access) [36].

O JNA é uma biblioteca de alto nível, cujo objetivo é permitir a um programa Java acessar bibliotecas escritas em C, porém não permite o acesso no sentido inverso, isto é, que um programa em C acesse uma biblioteca em Java [35], além de priorizar a facilidade de uso ao invés do desempenho [36].

O Java Native Interface (JNI) é uma interface de baixo nível, e é a fundação sobre a qual o JNA foi implementado. Permite fazer a interoperabilidade de programas escritos em linguagem Java com programas escritos em linguagem C de forma mais avançada, podendo carregar a máquina virtual e acessar métodos e objetos Java a partir de programas em C [40].

Essa interoperabilidade pode ser realizada de duas formas:

- O programa Java pode carregar, de modo dinâmico, uma biblioteca compartilhada (.so) escrita em C, e então executar suas funções.
- O programa C pode importar a *libjvm* e carregar uma JVM e então instanciar classes e invocar métodos Java - essa maneira de utilizar uma FFI é comumente chamada de “*embedding*”.

No entanto, a biblioteca C precisa ser escrita com suporte ao JNI, pois é necessário converter os dados de entrada (objetos) para uma forma com a qual o C possa trabalhar, e os resultados precisam ser formatados de forma adequada para que a JVM possa interpretá-los. Portanto, em geral, é desenvolvida uma “camada de tradução” para fazer a comunicação de um programa Java com bibliotecas C já existentes, ao invés de embutir tal tratamento nas próprias funções.

O JNI expõe diversos `typedefs` [33] correspondentes aos tipos de dado internos utilizados pela JVM, por exemplo:

- `jbyte` representa o tipo `byte` do java;
- `jobject` representa um objeto Java qualquer;
- `jbyteArray`, `jobjectArray`, ... representam um vetor do tipo especificado;
- ... entre outros.

Para a comunicação $C \rightarrow \text{Java}$ (C chamando métodos Java), o JNI fornece funções para trabalhar diretamente com os objetos Java. Por exemplo, para instanciar um objeto e chamar um de seus métodos é necessário [33]:

1. Buscar a classe do objeto, através da função `FindClass`.
2. Buscar o ID do construtor (i.e. um “ponteiro” para o construtor) através da função `GetMethodID`.
3. Converter os parâmetros para os tipos nativos Java. Por exemplo, se o construtor recebe como parâmetro uma *string*, é necessário construir um objeto `String` (seguindo estes mesmos passos 1, 2, e 3), convertendo a codificação dos caracteres de ASCII (utilizado no C) para Unicode (utilizado no Java).
4. Executar o construtor utilizando a função `CallObjectMethod`.
5. Repetir o procedimento 2, desta vez para obter o método que se deseja executar.
6. Executar o método desejado, da mesma forma que o procedimento 4.
7. Converter o resultado da chamada para um tipo de dado que se possa trabalhar no C. Por exemplo se o método retornou *string*, necessita-se de realizar a conversão do item 3 em ordem reversa.

Para a comunicação $\text{Java} \rightarrow C$ (Java chamando o C), é necessário criar, no Java, as assinaturas dos métodos a serem chamados no C, com a palavra chave `native` [33]. Já em C, deve-se criar as funções que se deseja chamar correspondentes às criadas no Java, recebendo e retornando tipos específicos do Java, e com uma assinatura específica [40].

Para tanto existe o utilitário `javah` que gera arquivos `.h` com essas assinaturas. Neste trabalho, de modo a isentar o usuário de lidar com as estruturas do JNI, são utilizadas funções intermediárias entre a função do usuário e a função JNI.

Após o carregamento da JVM, essas funções devem ser registradas junto à JVM, fornecendo-lhe os ponteiros para as funções. Dentro de cada uma delas, assim como no método anterior, devem ser feitas a conversão de tipos entre o C e o Java.

Outra preocupação importante na utilização do JNI é o gerenciamento de alocação dos objetos. Trabalhando com o código nativo corre-se o risco de não liberar objetos que não estão mais em utilização e ocasionar um vazamento de memória [33].

ctypes - a FFI da linguagem Python

Assim como a JNI, a `ctypes` funciona através de ligação dinâmica com uma biblioteca compartilhada (`.so`). Após o carregamento da biblioteca, é necessário indicar ao `ctypes` as assinaturas das funções que se deseja utilizar.

Ao contrário da JNI, porém, não é necessário criar funções em C com assinaturas específicas para essa interação—o `ctypes` é capaz de utilizar diretamente as funções de C. Isto é possível pois o `ctypes` não permite que a linguagem C utilize serviços da linguagem Python, então o código em C não precisa ser modificado para dar suporte às estruturas da linguagem Python [20].

2.4.2 Protocol Buffers

Em um sistema diversificado um grande desafio é a troca de informações entre diferentes implementações que, apesar de atender aos mesmos requisitos possuem formas distintas de representação dos dados, que precisam ser normalizados para possibilitar o intercâmbio entre quaisquer duas implementações.

O Protocol Buffers é uma biblioteca desenvolvida pela Google em 2008 [42], cujo objetivo é serializar de forma universal dados em qualquer linguagem de programação. Isto é alcançado através da implementação de compiladores que, através de uma descrição das estruturas de dados, geram código de serialização e deserialização em cada linguagem alvo, possibilitando que o programador trabalhe utilizando as estruturas de dados nativas de sua linguagem e a biblioteca faz todo o trabalho para convertê-la em uma representação única. Além disso, a implementação é flexível e permite que novas linguagens sejam suportadas, através da implementação do respectivo compilador.

A forma como a informação a ser serializada é estruturada deve ser especificada através da definição de tipos de mensagens em arquivos `.proto`. Cada mensagem possui um ou mais campos unicamente numerados, e cada campo possui um nome e um tipo. É

possível especificar campos como opcionais, obrigatórios, e repetidos (listas). Cada tipo de mensagem é um registro lógico contendo uma série de pares nome-valor, conforme o exemplo do Algoritmo 2 (adaptado de [23]).

Algoritmo 2 Definição de uma mensagem Protocol Buffers

```
1 message Pessoa {
2     required string nome = 1;
3     required int32 id = 2;
4     message Telefone {
5         required string number = 1;
6     }
7     repeated Telefone phone = 3;
8 }
```

No Algoritmo 2, é definida uma mensagem que transfere informações relativas a uma pessoa, que possui três atributos: um *id*, representado por um número inteiro, e um nome, representado por uma *string*. Ambos os campos são obrigatórios, sinalizado pela palavra-chave **required**. Além disso, uma lista de telefones é associada à pessoa. O Telefone é uma sub-entidade de Pessoa, e contém um único atributo, o número de telefone, representado por uma *string*. A palavra chave **repeated** sinaliza que aquele atributo é uma lista. À direita de cada campo, são especificados números que os identificam unicamente, chamados *tags*. *Tags* de 1 a 15 precisam de um *byte* para serem codificados, *tags* de 16 a 2047 precisam de dois *bytes*, etc. Deste modo, campos mais utilizados devem utilizar *tags* menores, para economizar espaço ao transmitir a mensagem. Campos não-obrigatórios e não utilizados não são transmitidos.

Feitas as definições das mensagens, é executado o compilador que gera o código de acesso a dados na linguagem especificada, com métodos *get/set* para cada campo, além de métodos para ler e serializar toda a estrutura para um vetor de *bytes*.

Algoritmo 3 Exemplo de utilização de Protocol Buffers em C++

```
1 Pessoa fulano;
2 fulano.set_name("Jose da Silva");
3 fulano.set_id(1234);
4 fulano.SerializeToOstream(&output);
```

O código do Algoritmo 3, em C++, exemplifica a utilização da biblioteca para transmitir uma mensagem. Para cada entidade da mensagem é criada uma classe, com métodos para atribuir valores a cada campo da mesma. No caso, é criada uma instância da classe Pessoa, seu nome e *id* são configurados, e por fim a mensagem é serializada e enviada através do *stream* “output”, que pode ser um arquivo, um soquete de rede, ou a saída do terminal.

Uma outra vantagem é que novos campos podem ser adicionados à mensagem sem quebra de compatibilidade – programas antigos simplesmente ignoram o novo campo quando é realizado o *parse* da mensagem.

Oficialmente são suportadas as linguagens C++ , Java e Python, e inúmeras outras foram implementadas pela comunidade em código-aberto [23].

2.5 Considerações

Apesar dos potenciais ganhos em segurança e confiabilidade ao se utilizar uma arquitetura de Replicação Máquina de Estados, essa ainda é pouco utilizada na prática. O objetivo do BFT-SMART é popularizar esse tipo de aplicação, fornecendo uma biblioteca robusta e bem-documentada, abstraindo a complexidade envolvida nesse tipo de protocolo [10]. Através da utilização de tecnologias estabelecidas como JNI e Protocol Buffers, este trabalho almeja a mesma facilidade para implementar aplicações diversificadas sobre o BFT-SMART. Além disso, como o Protocol Buffers funciona gerando código em cada linguagem de programação, têm-se mais um eixo de diversidade também na conversão da representação dos dados.

Capítulo 3

Arquitetura para Diversidade em Replicação Máquina de Estados

Neste capítulo será descrito em detalhe como foi implementada a diversidade no BFT-SMART, incluindo arquitetura do código, seus módulos e como cada um foi implementado. Conforme mencionado no Capítulo 1, a pretensão não é implementar todos os protocolos do BFT-SMART em diferentes linguagens, mas prover meios de implementar réplicas com diversidade utilizando esta biblioteca.

3.1 Princípios de projeto

Para implementar a diversidade no BFT-SMART, foram estabelecidas os seguintes princípios de desenvolvimento:

- Não deve ser necessário aos desenvolvedores conhecer todas as linguagens de programação para utilizar o BFT-SMART com diversidade (e.g. um programador C não precisa conhecer Java). É comum uma equipe ter profissionais para cada linguagem de programação, cada um deve implementar uma réplica na linguagem apropriada. Desta forma, é necessário fornecer interfaces para cada linguagem suportada e a arquitetura deve ser responsável por iniciar os protocolos do BFT-SMART.
- A execução de uma réplica ou cliente deve ser como é usual para cada linguagem e seu ambiente de execução. Consequentemente, deve ser evitada a utilização de dois processos (um em Java e outro em alguma outra linguagem) e mecanismos de comunicação inter-processual já que isto potencialmente impacta o desempenho geral do sistema.
- A interface entre o Java (BFT-SMART) e as outras linguagens de programação deve ser escrita em linguagem C, já que deve ser fácil a implementação de interface para

comunicação direta com outras linguagens, permitindo futura inclusão de suporte para outras linguagens de programação além das atuais.

Seguindo esses princípios, dois problemas devem ser resolvidos pela arquitetura proposta: (1) comunicação entre diferentes linguagens, i.e., é necessário que diferentes linguagens acessem métodos e funções de uma a outra; e (2) representação de dados, i.e., é necessário trocar dados entre diferentes linguagens, que podem utilizar diferentes representações dos dados.

Para resolver o problema (1), a arquitetura utiliza interfaces nativas fornecidas pelas linguagens de programação (FFIs) para permitir que o programa Java (linguagem do BFT-SMART) execute e/ou receba chamadas de programas C, e, a partir do C, para outras linguagens de programação. Para resolver o problema (2), será utilizado o Protocol Buffers, conforme descrito no capítulo anterior.

3.2 Modelo de programação

A solução proposta dá origem ao seguinte modelo de programação (Figura 3.1), que de certa forma se parece com o modelo RPC (*remote procedure call*—chamada de procedimento remoto):

1. Implementação de um arquivo Protocol Buffers com a representação dos dados. Este arquivo define os dados que serão trocados entre os processos através de mensagens de requisição/resposta. Este arquivo também deve definir os dados manipulados pelos protocolos de transferência de estado.
2. Compilação do arquivo Protocol Buffers para cada linguagem de programação utilizada no sistema, gerando todo o código auxiliar necessário para transformar os dados em um vetor de *bytes* e recuperar os dados de origem dele.
3. Implementação dos clientes e servidores, usando o código gerado anteriormente para troca de dados.
4. Compilação dos clientes e servidores, de acordo com cada linguagem e plataforma de destino, para gerar os códigos executáveis. Esta etapa é atrelada à linguagem e plataforma de destino, apresentando as diferenças entre elas. Por exemplo, um servidor em C pode ser compilado para executar no Windows enquanto um servidor Java é compilado para ser interpretado pela máquina virtual Java. Além disso, algumas linguagens interpretadas (e.g. Python) podem não necessitar de uma compilação.

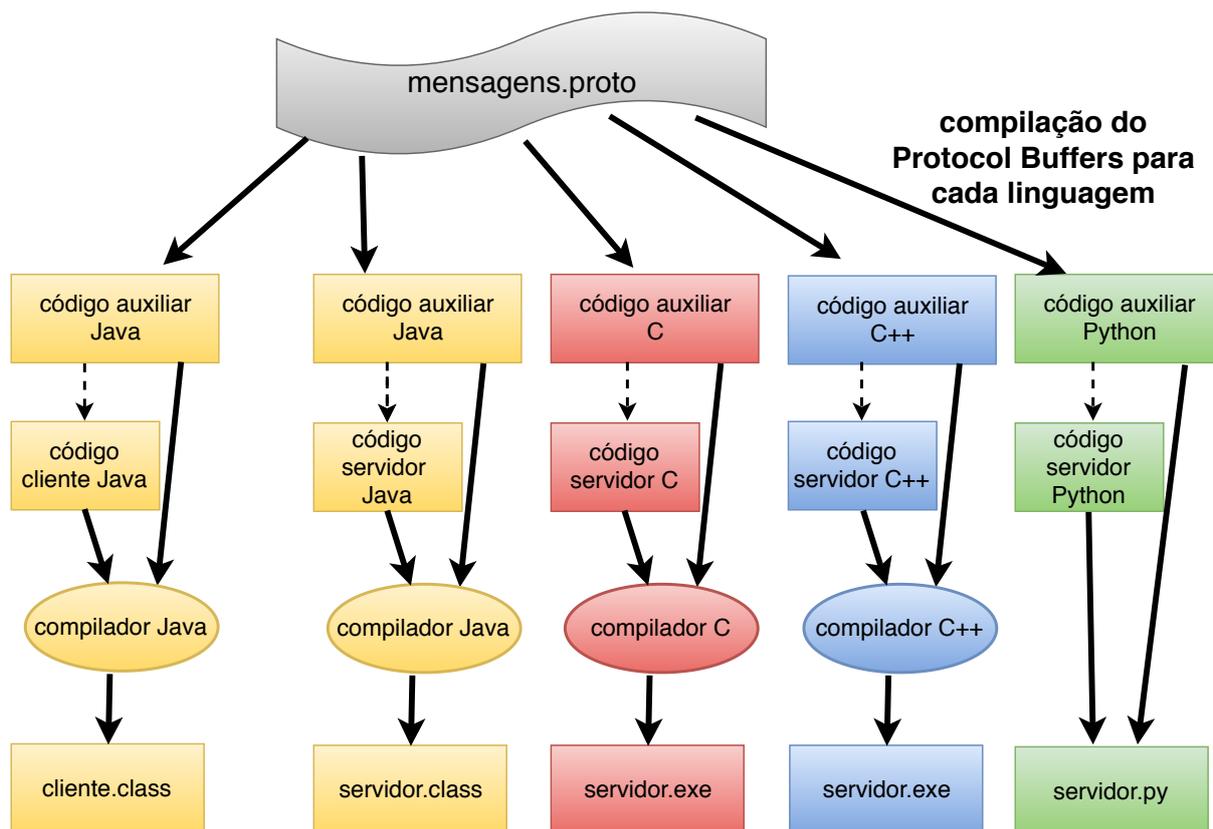


Figura 3.1: Modelo de programação.

3.3 Interface Bft-SMaRt

Esta seção detalha a interface escrita em C para comunicação com o BFT-SMaRt uma vez que foi utilizada para dar suporte a linguagens adicionais, e além disso, deve ser usada para fornecer suporte a outras linguagens além dessas já implementadas.

Para a comunicação entre as diferentes linguagens, a alternativa utilizada em nossa solução consiste em carregar a JVM (Máquina Virtual Java) no espaço de memória das réplicas (que podem ser escritas em outra linguagem de programação), fazendo chamadas diretas ao ambiente Java. Para tal foi utilizada a JNI (Java Native Interface), conforme apresentado na Seção 2.4.1. O BFT-SMaRt continua sendo um ponto em comum a todas as réplicas.

Alguns problemas potenciais com essa abordagem são:

1. *Conflitos entre a JVM e o ambiente de execução das outras linguagens.* Ambas operam sob o mesmo espaço de memória. Um exemplo de problema desta natureza, encontrado durante o desenvolvimento deste trabalho, é relacionado à interface com a linguagem Python: se houver alguma exceção não tratada a partir do Python, a JVM encerra com um *segfault*—violação de acesso à memória. A solução encontrada

neste caso foi a criação de um tratamento genérico para essas exceções que, quando acionado, finaliza a aplicação corretamente.

2. *Overhead introduzido pela cópia de buffers entre as camadas.* Via de regra, são necessárias cópias de *buffer* para converter a representação de vetores em uma linguagem para a representação em C, porém algumas linguagens podem oferecer otimizações desse processo. No entanto, conforme os experimentos do Capítulo 4, essas cópias não ofereceram impacto significativo no desempenho do sistema.

Para enfim integrar as réplicas em diversas linguagens com o BFT-SMaRT e intermediar todo o processo, foi proposta uma arquitetura em camadas, conforme a Figura 3.2. De modo a possibilitar a comunicação entre as diferentes linguagens e o Java, foi utilizada uma camada intermediária em linguagem C, visto que poucas linguagens oferecem suporte nativo a essa integração, e as que oferecem são linguagens já desenvolvidas sobre a JVM. Já para interface com a linguagem C, quase todas as linguagens de programação oferecem uma FFI para esse fim (conforme Capítulo 2).

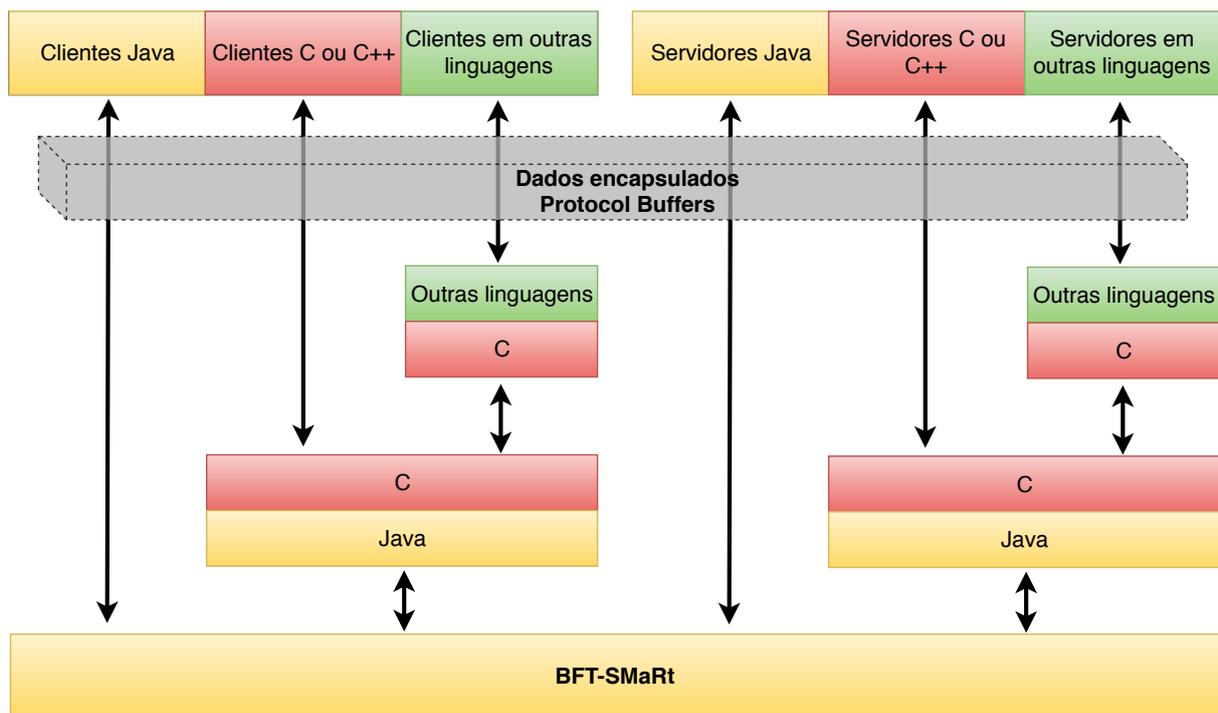


Figura 3.2: Camadas de interoperabilidade.

Para o envio de uma requisição do cliente ao servidor, cada camada é responsável pelas seguintes funcionalidades:

Cliente/servidor Essa é a parte escrita pelo usuário. Primeiramente as estruturas de dados são descritas na linguagem do Protocol Buffers (arquivo `.proto`), o qual é

compilado e gera o código especializado de serialização e deserialização. Apenas vetores de *byte* são repassados entre as camadas.

Interface C ↔ outras linguagens Esta camada irá abstrair a conversão de estruturas de dados do C para as estruturas específicas das outras linguagens, e vice-versa. Essa camada permitirá que se implemente somente uma vez a conversão de tipos C/Java, que é a parte mais trabalhosa da comunicação, além de fornecer uma interface idiomática na linguagem de destino.

Interface Java ↔ C Realiza o tratamento das informações recebidas para que possam ser interpretados pela JVM. Esta camada recebe as informações da camada em C e comunica-se com o BFT-SMART.

No caso do recebimento das requisições pelo servidor e do recebimento da resposta pelo cliente, as camadas seguem a lógica inversa.

3.3.1 Camada Java ↔ C

A camada de interface Java ↔ C tem como objetivos o carregamento da JVM dentro do programa que a utiliza, a tradução e conversão das estruturas e tipos de dados nativos da linguagem C para as estruturas e tipos de dados Java e a chamada de funções e métodos, tanto no sentido C → Java (no caso do cliente), quanto no sentido Java → C (no caso do servidor).

As camadas em si foram implementadas utilizando funcionalidades do C++, por facilitar significativamente o trabalho com o JNI, e devem ser compiladas em um compilador apropriado. No entanto, a interface exposta por elas é utilizável a partir da linguagem C, pois foi empregada a cláusula `extern C`.

Interface de cliente

A interface de cliente possibilita a implementação de um cliente, em linguagem C, que irá utilizar um serviço implementado no BFT-SMART. Do lado Java foi criada a classe `ClientWrapper` (Algoritmo 4), que fornece uma interface simplificada para a classe `ServiceProxy` do BFT-SMART, de modo a facilitar a chamada a partir do C.

As operações e tipos de dados expostos pela interface de cliente (Algoritmo 5) são as seguintes:

BFT_BYTE corresponde ao tipo nativo `byte` em Java, que em C equivale a `signed char`.

Algoritmo 4 Classe Java `ClientWrapper`

```
1 public class ClientWrapper {
2     private ServiceProxy proxy;
3     public ClientWrapper(int id) {
4         proxy = new ServiceProxy(id);
5     }
6     public byte[] executeOrdered(byte[] request) {
7         return this.proxy.invokeOrdered(request);
8     }
9     public byte[] executeUnordered(byte[] request) {
10        return this.proxy.invokeUnordered(request);
11    }
12 }
```

setClasspath utilizada caso o usuário deseje utilizar um *classpath* customizado no momento de carregamento da JVM.

invokeOrdered realiza uma chamada à função `executeOrdered` do BFT-SMART.

invokeUnordered realiza uma chamada à função `executeUnordered` do BFT-SMART.

finalizeJVM aguarda pelo término de todas as *threads* em execução na JVM e então a descarrega da memória.

loadJVM carrega a JVM no espaço de memória do programa.

createServiceProxy cria uma instância da classe `ClientWrapper`, que conecta aos servidores e fica aguardando chamadas.

Algoritmo 5 Funções expostas para clientes em C

```
1 typedef signed char BFT_BYTE;
2 void setClasspath(const char* cp);
3 int createServiceProxy(int id);
4 int invokeOrdered
5     (BFT_BYTE command[], int tamanho, BFT_BYTE saida[]);
6 int invokeUnordered
7     (BFT_BYTE command[], int tamanho, BFT_BYTE saida[]);
8 void finalizeJVM();
9 int loadJVM();
```

As funções `invokeOrdered` e `invokeUnordered` no BFT-SMART recebem e retornam vetores de *byte*. Para implementar essa estrutura em C, foi seguida a convenção utilizada em funções como a `fread` e a `fwrite` – para a entrada de dados, a função recebe um ponteiro para o vetor de entrada e um inteiro com a quantidade de elementos a utilizar

deste vetor. Para a saída de dados, a função recebe um ponteiro para o vetor de saída, que já deve estar alocado, e retorna um inteiro, que corresponde à quantidade de *bytes* que foram escritos no *buffer*. Quando o usuário não desejar mais realizar chamadas, deve ser chamada a função `finalizeJVM`, para que a JVM possa ser finalizada de modo correto, com a finalização das conexões com as outras instâncias do BFT-SMART. Caso o usuário retorne da função `main` sem finalizar a JVM, a mesma será encerrada abruptamente, sem oportunidade de realizar mais procedimentos.

Interface de servidor

Para implementação de uma réplica servidor BFT, o BFT-SMART possibilita o desenvolvimento de uma implementação de uma classe abstrata, `DefaultSingleRecoverable`. Para uma comunicação Java \leftrightarrow C que possibilite a implementação de réplicas em outras linguagens, foi desenvolvida uma implementação mínima dos 4 métodos requeridos pela classe `DefaultSingleRecoverable`, através da classe `ServerWrapper` (Algoritmo 6), cada um apontando para um método “nativo” que, em tempo de execução, é associado a uma função em código C.

Algoritmo 6 Métodos nativos da classe `ServerWrapper`

```
1 public class ServerWrapper extends DefaultSingleRecoverable {
2     private native byte[] executeOrderedNative(byte[] command);
3     private native byte[] executeUnorderedNative(byte[] command);
4     private native byte[] getSnapshotNative();
5     private native void installSnapshotNative(byte[] command);
6     @Override
7     public byte[] appExecuteOrdered(byte[] command,
8         MessageContext msgCtx) {
9         return executeOrderedNative(command);
10    }
11    @Override
12    public byte[] executeUnordered(byte[] command,
13        MessageContext msgCtx) {
14        return executeUnorderedNative(command);
15    }
16    @Override
17    public void installSnapshot(byte[] state) {
18        installSnapshotNative(state);
19    }
20    @Override
21    public byte[] getSnapshot() {
22        return getSnapshotNative();
23    }
24 }
```

Para a implementação da camada Java \leftrightarrow C de servidor, também foi adotado o C++ com a exposição da interface pública em C, como descrito anteriormente para o caso do cliente.

As operações e tipos de dados expostos pela interface de servidor (Algoritmo 7) são as seguintes:

implement* recebem como parâmetro um ponteiro para função que contém a implementação completa das respectivas operações. O usuário da biblioteca deve obrigatoriamente chamar essas funções antes de iniciar a réplica, pois em tempo de compilação a biblioteca de diversidade não possui referências para as funções que o usuário irá implementar futuramente. Portanto é necessário o “registro” das mesmas através de *callbacks*, prática comum em programas C [41]. Por exemplo, se a função que implementa as operações ordenadas em C se chamasse `execOrd`, então na inicialização deve ser executada a função `implementExecuteOrdered(&execOrd)`.

startServiceReplica inicia a réplica, conectando às outras instâncias e podendo receber conexões de clientes. A partir deste momento, as funções fornecidas previamente através de **implement*** serão invocadas pelo BFT-SMART.

Algoritmo 7 Interface exposta para servidores em C

```
1 void implementExecuteOrdered
2   (int (*impl) (BFT_BYTE [], int, BFT_BYTE []));
3 void implementExecuteUnordered
4   (int (*impl) (BFT_BYTE [], int, BFT_BYTE []));
5 void implementInstallSnapshot(void (*impl) (BFT_BYTE [], int));
6 void implementgetSnapshot(int (*impl) (BFT_BYTE []));
7 int startServiceReplica(int id);
8 void finalizeJVM();
9 int loadJVM();
```

3.3.2 Interface C \leftrightarrow outras linguagens

Para C e C++, a interface Java \leftrightarrow C pode ser utilizada diretamente, sem necessidade de camada intermediária, pois as linguagens são totalmente compatíveis. Para a utilização com outras linguagens de programação, deve ser criada uma camada adicional que utilize os serviços da camada Java \leftrightarrow C e exponha esse serviço para a linguagem alvo. Neste trabalho foi criada uma camada para a linguagem Python, utilizando a interface `ctypes`. Cada linguagem precisará de uma implementação distinta, no entanto a forma como são utilizados os serviços da camada Java \leftrightarrow C será constante.

Como exemplo, discutiremos como foi desenvolvida a camada intermediária para Python. A utilização da interface Python é simples e o código se resumiu a implementação de duas classes:

BFTJVM classe abstrata, responsável por inicializar a JVM e configurar a assinatura das funções C (parâmetros e tipo de retorno). Uma listagem resumida desta classe é mostrada no Algoritmo 22.

BFTSMaRtServer classe abstrata, herdada da classe **BFTJVM**, que torna idiomática¹ a implementação das réplicas. Ao invés de lidar diretamente com as funções de C, o programador herda dessa classe implementando os métodos abstratos `executeOrdered` e `executeUnordered`, de forma análoga a uma implementação Java puro. Uma listagem resumida da classe **BFTSMaRtServer** é mostrada no Algoritmo 8.

Ao ser instanciada, a classe **BFTJVM**:

1. Carrega a biblioteca compartilhada *libbftsmr.so*, que consiste na camada Java ↔ C.
2. Configura o tipo de retorno das funções, através da propriedade `restype`. Não é necessário configurar os parâmetros das funções, que serão resolvidos dinamicamente a depender da maneira de como as funções forem chamadas.
3. Cria tipos que representam as funções de *callback* que serão chamadas pela camada em C, como por exemplo a `executeOrdered`, que será associada ao tipo `INVOKORDFUNC`, que retorna um inteiro (`c_int`) e recebe, nesta ordem, um ponteiro para char, um inteiro, e um ponteiro para ponteiro para char.

As funções `bftsmartallocate` e `bftsmartrelease`, definidas na camada em C, possibilitam ao Python alocar vetores em C (internamente utilizando `malloc`), já que Python não possibilita gerenciamento manual da memória.

A classe **BFTSMaRtServer** (Algoritmo 8) utiliza os tipos definidos em **BFTJVM** (Algoritmo 22) para encapsular funções Python em ponteiros para função C, estes em seguida são repassados ao código em C através da chamada `implementExecuteOrdered`.

Cada uma das funções que é chamada a partir do C realiza as seguintes operações:

1. Cria um *buffer* no espaço Python (`create_string_buffer`), e copia para ele o vetor de entrada (que vem do C).
2. Executa a função `invokeOrdered` do usuário.

¹Um idioma é um meio padrão de se representar uma estrutura recorrente ou se resolver um problema em uma linguagem de programação específica, em geral utilizando funcionalidades específicas daquela linguagem [1]. Nesse contexto, uma implementação idiomática é aquela que utiliza plenamente as funcionalidades da linguagem de programação utilizada.

Algoritmo 8 Trecho da classe BFTSMaRtServer

```
1 # herda da classe BFTJVM
2 class BFTSMaRtServer(BFTJVM):
3 # parametro id: o id da replica como configurado no bft-smart.
4 def __init__( self , clspath , id , dllpath ):
5     super(BFTSMaRtServer,self).__init__(dllpath,clspath)
6     # função que será chamada pelo C.Trata parametros
7     # de funcao, retorno e conversao de tipos, e chama a
8     # funcao da aplicacao, implementada pelo usuario.
9     def intermediateInvokeOrdered(input, size, outmem):
10        # recebe a requisição da camada C
11        p = create_string_buffer ( size )
12        memmove(p, input, size)
13        result = self.invokeOrdered(p) # chama a função do usuário
14        # converte e envia a resposta à camada C
15        output = BFTJVM.libbft.bftsmartallocate(len(result))
16        memmove(output, result, len(result))
17        outmem[0] = cast(output, POINTER(c_char));
18        return len( result )
19    # gera um ponteiro para função para intermediateInvokeOrdered.
20    self .invokeOrdCback = INVOKORDFUNC(intermediateInvokeOrdered)
21    # registra esse ponteiro para a camada C chamar posteriormente.
22    BFTJVM.libbft.implementExecuteOrdered(self.invokeOrdCback)
23    # inicia a réplica
24    BFTJVM.libbft.startServiceReplica(int(id))
```

3. Aloca um vetor de C, e copia o resultado do processamento da `invokeOrdered` (que vem em um *buffer* no espaço Python) para ele.
4. Retorna, por referência, o ponteiro para o vetor que contém o resultado.
5. Retorna o tamanho do vetor de resposta, possibilitando ao C percorrê-lo.

Por último, o construtor inicia o funcionamento da réplica, através da função `startServiceReplica`.

3.3.3 Cliente/Servidor em outra linguagem

As etapas gerais para implementar uma aplicação diversificada utilizando a solução proposta neste trabalho são:

1. Projetar as mensagens que serão trocadas, através da criação dos arquivos `.proto`.
2. Compilar os arquivos `.proto`, gerando as classes de interface.
3. Programar a aplicação (cliente e servidor), nas linguagens escolhidas, utilizando as classes geradas no item 2.
4. Executar as réplicas e o cliente, passando os parâmetros apropriados.

A seguir descreve-se de forma detalhada a implementação de clientes e servidores em cada uma das linguagens implementadas neste trabalho. Um exemplo de implementação com trechos de código é apresentada no Apêndice A.

Java

Protocol Buffers. Primeiramente é necessário compilar os arquivos `.proto` (Seção 2.4.2) do Protocol Buffers para gerar as classes que irão converter as mensagens em objetos Java e vice-versa. Para tal utiliza-se o comando:

```
protoc --java_out=diretorio_saida Mensagem.proto
```

As classes serão gravadas no diretório `diretorio_saida`, e podem ser compiladas como parte do projeto principal.

Servidor. Em sua forma mais simples, o servidor em Java é implementado através de herança da classe `DefaultSingleRecoverable`, que possui os métodos abstratos `executeOrdered`, `executeUnordered`, `getSnapshot` e `installSnapshot`, que devem ser implementados conforme a Seção 2.2.1 e [10].

Cliente. No programa cliente, deve ser instanciada a classe `ServiceProxy`, a qual fornece os métodos `invokeOrdered` e `invokeUnordered`. Esses métodos devem ser executados fornecendo o vetor de *bytes* proveniente do Protocol Buffers.

C

Protocol Buffers. Para gerar os arquivos `.c` e `.h` que fornecem as funções de conversão das mensagens em *structs* C e vice-versa. Para tal utiliza-se o comando:

```
protoc-c --c_out=diretorio_saida Mensagem.proto
```

Os arquivos gerados serão gravados no diretório `diretorio_saida`, e podem ser compilados como parte do projeto principal.

Servidor. Para implementar o servidor, é necessário o registro dos callbacks para as funções `executeOrdered` e `executeUnordered`. Após o registro, carrega-se a JVM através da função `loadJVM`, e executando a função `startServiceReplica` o servidor entra em operação.

Cliente. Para o cliente em C, é necessário carregar a JVM, após iniciar o *proxy* através da função `createServiceProxy`, e então pode-se utilizar as funções `invokeOrdered` e `invokeUnordered` para realizar as chamadas aos servidores.

C++

Protocol Buffers. Para gerar os arquivos `.cc` e `.h` que fornecem as funções de conversão das mensagens em objetos C++ e vice-versa. Para tal utiliza-se o comando:

```
protoc --cpp_out=diretorio_saida Mensagem.proto
```

Os arquivos gerados serão gravados no diretório `diretorio_saida`, e podem ser compilados como parte do projeto principal.

Servidor. Para implementar o servidor em C++, é utilizada a mesma biblioteca da implementação em C, portanto também é necessário o registro dos *callbacks* para as funções `executeOrdered` e `executeUnordered`. Após o registro, carrega-se a JVM através da função `loadJVM`, e executando a função `startServiceReplica` o servidor entra em operação. A diferença está na utilização do Protocol Buffers.

Cliente. Assim como para o cliente em C, é necessário carregar a JVM, após iniciar o *proxy* através da função `createServiceProxy`, e então pode-se utilizar as funções `invokeOrdered` e `invokeUnordered` para realizar as chamadas aos servidores.

Phyton

Protocol Buffers. Para gerar os arquivos `.py` que fornecem as funções de conversão das mensagens em objetos Python e vice-versa. Para tal utiliza-se o comando:

```
protoc --python_out=diretorio_saida Mensagem.proto
```

Os arquivos gerados serão gravados no diretório `diretorio_saida`, e podem ser executados como parte do projeto principal.

Servidor. Para utilizar o servidor em Python, utiliza-se herança da classe `BFTSMaRtServer`, e implementar os métodos abstratos assim como na implementação em Java. Ao instanciar essa classe, o servidor entra em execução, assim como na implementação em Java.

Cliente. Para utilizar o cliente em Python, instanciar a classe `BFTSMaRtClient`, e chamar seus métodos `invokeOrdered` e `invokeUnordered` de acordo com a necessidade.

Go

Protocol Buffers. Para gerar os arquivos `.go` que fornecem as funções de conversão das mensagens em objetos Go e vice-versa (utilizando o *plugin* `protoc-gen-go`). Para tal utiliza-se o comando:

```
protoc --plugin=protoc-gen-go=/usr/bin/protoc-gen-go
      --go_out=diretorio_saida Mensagem.proto
```

Os arquivos gerados serão gravados no diretório `diretorio_saida`, e podem ser compilados como parte do projeto principal.

Servidor. Para utilizar o servidor em Go, deve-se criar um tipo que forneça uma interface com os métodos `ExecuteOrdered` e `ExecuteUnordered`. Deve, então, ser invocada a função `StartServiceReplica` do módulo `bftsmartserver`, fornecendo uma referência para uma instância do tipo criado. Ao chamar essa função o servidor entra em execução.

Cliente. Para utilizar o cliente em Go, deve-se iniciar o *proxy* através da função `CreateServiceProxy` do módulo `bftsmartclient`, e então pode-se utilizar as funções `InvokeOrdered` e `InvokeUnordered` para realizar as chamadas aos servidores.

3.4 Considerações Finais

Este capítulo apresentou nossa proposta de uma arquitetura para possibilitar a implementação de diversidade em serviços replicados através de Replicação Máquina de Estados. Desta forma, é possível incluir propriedades de segurança e dependabilidade em serviços ou aplicações implementados segundo esta abordagem. Um exemplo de como essa arquitetura pode ser empregada para desenvolver uma aplicação pode ser encontrado no Apêndice **A**. Além disso, todas as implementações desenvolvidas estão disponíveis como código-aberto no seguinte repositório: <https://github.com/bftsmart-project/diversity>.

Capítulo 4

Experimentos

Visando analisar o desempenho da arquitetura proposta e da implementação desenvolvida, bem como o comportamento de uma Replicação Máquina de Estados com diversidade, alguns experimentos foram realizados no Emulab [44], que é uma plataforma que possibilita configurar ambientes e fornece máquinas físicas e virtuais, sem custo, para execução de testes e experimentos. Neste capítulo são descritos os experimentos realizados, seu ambiente de execução e os resultados coletados.

4.1 Aplicação

Foram desenvolvidos três testes: um teste nulo (requisição e resposta vazios) para avaliar o *overhead* da implementação, uma aplicação de lista foi desenvolvida para ter seu desempenho analisado—suas implementações estão descritas no Apêndice A— e um teste com transferência de vetores de *bytes* de diversos tamanhos para avaliar o impacto das conversões entre as linguagens.

Cada servidor executou em uma máquina separada, enquanto que 100 clientes executaram na outra máquina. O experimento de lista teve uma fase de *warm-up* que consistiu na inicialização da lista com 200k operações **ADD**—com exceção do *warm-up* da própria operação de **ADD**, que consistiu na execução aleatória de operações dos outros tipos, e nesse caso o *warm-up* executa 10% do número total de operações **ADD** executadas.

Em nossos experimentos, a lista foi inicializada com 200k entradas em cada réplica (2k entradas por cliente) e os valores para as operações **ADD**, **GET** e **REMOVE** foram selecionados aleatoriamente seguindo uma distribuição uniforme, com valores de 0 a 400k. Além disso, todos os clientes executados foram desenvolvidos na linguagem Java, enquanto que os servidores foram diversificados conforme os cenários descritos nas seções seguintes.

A latência e o *throughput* destas operações foram determinados, com o objetivo de comparar 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 [10] a cada 5000 requisições.

4.2 Análise do Desempenho: Sem Diversidade no Ambiente de Execução

Primeiramente, executamos alguns experimentos em um ambiente onde todas as máquinas possuíam a mesma configuração, o que possibilitou uma análise mais precisa a respeito do *overhead* introduzido pelo uso de diversidade. Desta forma, o ambiente foi consistindo por 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). O BFT-SMART foi configurado com $n = 4$ servidores para tolerar até uma falha Bizantina.

4.2.1 Resultados e Análises

A Figura 4.1 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.

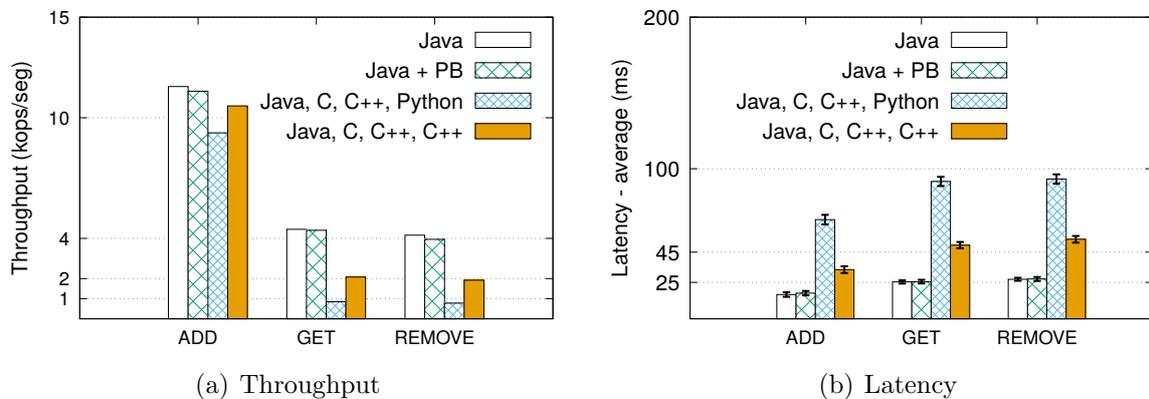


Figura 4.1: Experimentos sem diversidade no ambiente de execução.

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, devido às variações de desempenho específicas à cada uma delas. Este comportamento 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 operações ADD, GET e REMOVE). Como podemos observar na Tabela 4.1, a implementação em Python apresentou um tempo de resposta muito superior às demais, enquanto C e C++ tiveram latências próximas, mas que representam 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 possui desempenho mais baixo, ficando muito penalizada em relação às demais, o BFT-SMART iniciava os protocolos para transferência e atualização de estados [10]. 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 Figura 4.1 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 das operações demorarem praticamente o dobro em C++ (Tabela 4.1). 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)	15.01	52.40	30.16	90.83
	Desvio Padrão (ms)	0.09	0.17	0.13	1.45
GET	Tempo de execução (ms)	22.63	66.10	47.71	106.82
	Desvio Padrão (ms)	0.10	0.32	0.10	1.36
REMOVE	Tempo de execução (ms)	25.37	69.19	52.03	113.42
	Desvio Padrão (ms)	0.06	0.19	0.12	0.91

Tabela 4.1: Tempo de resposta para cada linguagem (execução do `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—sem utilização de Protocol Buffers). Este tipo de aplicação é comumente utilizado para avaliar estes sistemas [10] e apenas os tamanhos das requisições/respostas são configurados. A Tabela 4.2 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, evidenciando que o *overhead* da implementação é negligível.

	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 4.2: Experimento para um aplicação vazia (0/0).

4.3 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.

4.3.1 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 uma mesma vulnerabilidade não esteja presente em mais de um deles. A Tabela 4.3 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 [44]. 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 (Capítulo 3).

Eixo de Diversidade		Réplica 1	Réplica 2	Réplica 3	Réplica 4
Aplicação		Java	C++	C	Python
Sistema Operacional		Fedora Core 15 <i>kernel 2.6.40</i>	FreeBSD 10.0	CentOS 7.1 <i>kernel 3.10.0</i>	Ubuntu 12.04 <i>kernel 3.2.0</i>
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 4.3: 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.

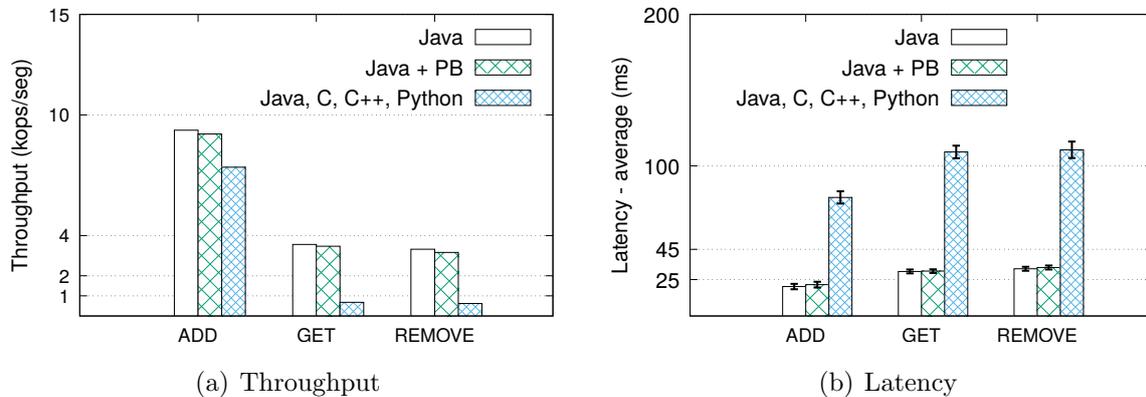


Figura 4.2: Experimentos com diversidade no ambiente de execução.

4.3.2 Resultados e Análises.

A Figura 4.2 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. Por fim, é importante acrescentar que neste caso, devido a utilização de máquinas com menor poder de processamento [44], o desempenho ficou reduzido em comparação aos experimentos anteriores.

4.4 Análise Complementar do Desempenho

Nos experimentos anteriores foram necessárias quatro réplicas para tolerar até uma falha, e as linguagens escolhidas para análise foram Java, C, C++ e Python. Nesta seção faremos uma análise mais aprofundada a respeito do desempenho da arquitetura proposta, considerando todas as linguagens atualmente suportadas, i.e., incluindo também a linguagem Go.

Para isso, analisaremos o sistema quando configurado com cada uma das linguagens em uma aplicação vazia, *i.e.*, sem lógica de aplicação, transmitindo apenas dados de tamanho pré-definidos, de modo a testar o *overhead* introduzido no sistema por cada implementação ou linguagem. Neste experimento, as requisições e respostas consistem de vetores de *bytes*, inicializados com valores fixos. Utilizamos tamanhos de 0, 256, 512, 1k e 4k para requisições e respostas. Os vetores são encapsulado em estruturas Protocol Buffers.

O ambiente consistiu de 5 máquinas d430 (2 processadores Intel E5-2630v3 CPUs a 2.4 GHz (8 cores), 64GB de RAM e interface de rede Gigabit) conectadas a um *switch* de 1Gbps. O software utilizado foi o sistema operacional Ubuntu 18.04. O BFT-SMARTfoi configurado com $n = 4$ servidores para tolerar até uma falha. Todos os servidores foram implementados na mesma linguagem, enquanto que os clientes foram implementados sempre em Java.

Novamente cada servidor executou em uma máquina separada, enquanto que 100 clientes executaram em outra máquina. O experimento consistiu no envio de 200k requisições (as respostas possuíam o mesmo tamanho que as requisições) e a latência e o *throughput* foram obtidos para análise.

4.4.1 Resultados e Análises

A Figura 4.3 apresenta os valores obtidos na execução deste experimento. Novamente todas as aplicações obtiveram resultado similar, à exceção da linguagem Python, novamente por possuir desempenho mais baixo. Também foi possível observar que, à medida que o tamanho das requisições aumenta, o maior gargalo no desempenho do sistema está relacionado com os dados trafegados na rede, o que evidencia que as cópias em memória dessa informação não exerce impacto significativo no desempenho.

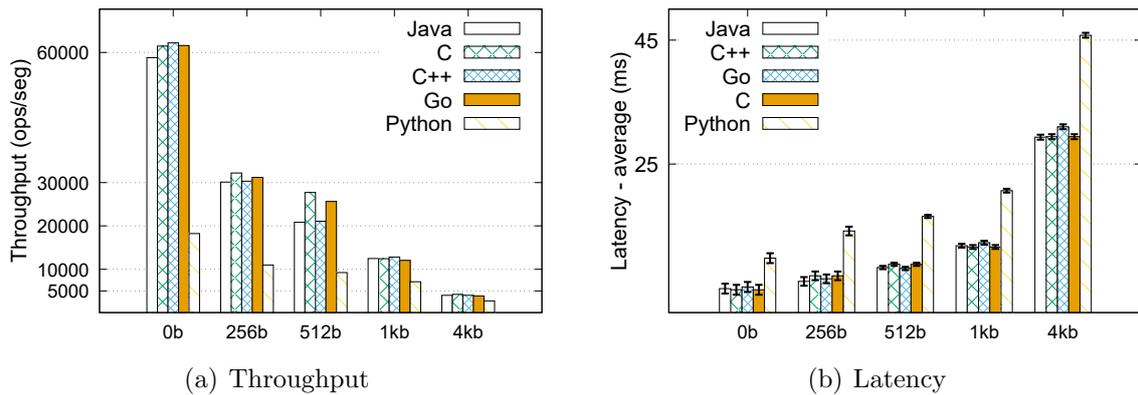


Figura 4.3: Experimentos com requisições e respostas de diferentes tamanhos.

4.5 Considerações Finais

Através dos experimentos apresentados neste trabalho (principalmente com a implementação vazia), conclui-se que a implementação proposta possui *overhead* mínimo sobre o desempenho da aplicação. Além disso, foram realizadas algumas observações importantes, durante a avaliação experimental:

1. *Custos de implementar réplicas em várias linguagens de programação versus segurança.* Claramente, os custos para implementar uma aplicação em várias linguagens de programação é maior do que ter apenas uma implementação para ela. Porém, muitos sistemas críticos devem ter uma fortes propriedades de segurança e dependabilidade, e, conseqüentemente, esses custos são justificados.
2. *Distribuição e gerenciamento de uma aplicação diversificada podem ser complexos.* A distribuição e o gerenciamento de um sistema diversificado podem ser complexos, já que é necessário configurar vários ambientes de execução utilizando diferentes sistemas operacionais, bibliotecas, administradores, etc. A Tabela 4.3 dá um bom exemplo de tal complexidade.
3. *O desempenho do sistema como um todo é limitada pela réplica com o pior desempenho.* Já que as réplicas devem evoluir de maneira sincronizada (realizando os mesmos passos) e clientes devem aguardar por um quorum de respostas idênticas para completar uma requisição ($f + 1$ respostas), o sistema irá executar na velocidade das réplicas mais lentas.

Capítulo 5

Conclusões

Este capítulo apresenta uma visão geral daquilo que foi desenvolvido, lembrando as principais contribuições desta dissertação e como as mesmas foram alcançadas. Por fim, as perspectivas para continuidade do projeto são apresentadas.

5.1 Visão Geral do Trabalho

Este trabalho apresentou uma nova abordagem para RME que utiliza diversidade na implementação das réplicas para aumentar a segurança das aplicações. De fato, é muito provável que uma mesma vulnerabilidade não possa ser explorada em mais de uma linguagem de programação, i.e., cada linguagem e implementação apresentará suas próprias vulnerabilidades. Além disso, a solução proposta permite a integração dos protocolos de uma RME em sistemas legados escritos em outras linguagens de programação. Com isso, é possível tanto implementar serviços novos quanto adicionar suporte a dependabilidade e segurança em sistemas legados.

A arquitetura proposta para diversidade foi implementada e integrada no BFT-SMART. Vale destacar que o objetivo deste trabalho não é implementar diferentes versões de uma RME em diferentes linguagens de programação, mas sim utilizar alguma implementação já existente, neste caso o BFT-SMART. Através de um conjunto de experimentos foi possível analisar o comportamento prático de uma RME com diversidade. De uma forma geral, como o sistema evolui de forma sincronizada, o desempenho total do sistema é limitado pelo desempenho da réplica mais lenta dentre as utilizadas. No entanto, esta abordagem aumenta o nível de segurança no sistema, um requisito primordial para muitas aplicações.

5.2 Revisão dos Objetivos e Contribuições

Cada um dos objetivos específicos deste trabalho foram alcançados conforme discutido a seguir:

1. **Estudar os conceitos envolvendo diversidade e RME.** Esse objetivo foi alcançado através de uma revisão de bibliografia e revisão de estado da arte, que embasam essa monografia, conforme discutido no Capítulo 2.
2. **Proposta de uma arquitetura para suportar diversidade na implementação de réplicas de uma RME.** Esse objetivo foi atingido através da arquitetura proposta na Seção 3.3, e devido à utilização de camadas intermediárias em C com utilização de JNI (Seção 2.4.1) e Protocol Buffers (Seção 2.4.2), é extensível e com bom desempenho, como evidenciado nos experimentos do Capítulo 4.
3. **Integração e descrição de como esta arquitetura foi integrada no Bft-SMaRt.** A arquitetura foi implementada e integrada no BFT-SMART, além disso foram implementadas 4 linguagens de programação além do Java (Go, C, C++ e Python), como descrito na Seção 3.3. Também foram descritos os passos para implementação de uma aplicação com diversidade no Anexo A.
4. **Apresentação e análise de experimentos com as implementações realizadas.** Por fim, este objetivo foi alcançado através dos experimentos descritos no Capítulo 4, concluindo que a implementação é viável e apresenta bom desempenho, analisando o comportamento do sistema quanto à diferença de desempenho das diferentes linguagens de programação, e observando como evitar problemas de desempenho decorrentes disso.

Todas as implementações desenvolvidas estão disponíveis como código-aberto no seguinte repositório: <https://github.com/bftsmart-project/diversity>. Além disso, algumas contribuições desta dissertação foram publicadas em conferências e periódicos [16, 17, 19].

5.3 Perspectivas Futuras

Como trabalhos futuros, pretende-se implementar suporte para outras linguagens de programação, além de explorar outras aplicações com o intuito de aumentar a compreensão sobre o funcionamento de uma Replicação Máquina de Estados com diversidade. Além disso, também pretendemos analisar os efeitos da utilização de diversidade nos clientes, embora este não seja o foco principal do trabalho, e realizar mais experimentos variando

principalmente a localização do líder (processo mais sobrecarregado durante a execução da RME, pois é responsável pela elaboração das propostas de ordenação do algoritmo de difusão atômica [10, 11]), analisando características de *hardware* e de outras linguagens.

Finalmente, a arquitetura proposta neste trabalho, principalmente pela adição de suporte para a linguagem de programação Go, possibilitará que o BFT-SMART seja integrado de forma direta na plataforma *Hyperledger Fabric Blockchain* [4]. A integração atual emprega outros artifícios, como comunicação através de *sockets*, para utilizar o BFT-SMART como a camada de ordenação desta plataforma [39].

Referências

- [1] Harold Abelson and Gerald Jay Sussman. *Structure and interpretation of computer programs*. MIT PRESS, 1983. 26
- [2] Eduardo Adilio Pelinson Alchieri, Alysson Neves Bessani, and Joni da Silva Fraga. Replicação máquina de estados dinâmica. In *Anais do XIV Workshop de Teste e Tolerância a Falhas- WTF 2013*, 2013. 8
- [3] Yair Amir, Brian Coan, Jonathan Kirsch, and John Lane. Prime: Byzantine replication under attack. *IEEE Transactions on Dependable and Secure Computing*, 8(4):564–577, 2011. 1
- [4] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. Hyperledger fabric: A distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 30:1–30:15, New York, NY, USA, 2018. ACM. 40
- [5] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, March 2004. 1, 5
- [6] A. Avizienis and L. Chen. On the implementation of n-version programming for software fault tolerance during execution. In *International Computer Software and Applications Conference*, 1977. 10
- [7] Eli Barzilay and Dmitry Orlovsky. Foreign interface for plt scheme. *on Scheme and Functional Programming*, page 63, 2004. 13
- [8] Abhijit Belapurkar, Anirban Chakrabarti, Harigopal Ponnappalli, Niranjan Varadarajan, Srinivas Padmanabhuni, and Srikanth Sundarrajan. *Distributed systems security: issues, processes and solutions*. John Wiley & Sons, 2009. 4, 5
- [9] Alysson Bessani, Alessandro Daidone, Ilir Gashi, Rafael Obelheiro, Paulo Sousa, and Vladimir Stankovic. Enhancing fault / intrusion tolerance through design and configuration diversity. In *Proceedings of the 3rd Workshop on Recent Advances on Intrusion-Tolerant Systems*, 2009. 1, 10

- [10] Alysson Bessani, João Sousa, and Eduardo Alchieri. State machine replication for the masses with BFT-SMaRt. In *Proc. of the International Conference on Dependable Systems and Networks*, 2014. [1](#), [2](#), [3](#), [8](#), [10](#), [17](#), [28](#), [32](#), [33](#), [34](#), [40](#)
- [11] Miguel Castro and Barbara Liskov. Practical Byzantine fault-tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, November 2002. [1](#), [8](#), [40](#)
- [12] Miguel Castro, Rodrigo Rodrigues, and Barbara Liskov. Base: Using abstraction to improve fault tolerance. *ACM Transactions on Computer Systems (TOCS)*, 21(3):236–269, 2003. [3](#), [12](#)
- [13] Liming Chen and Algirdas Avizienis. N-version programming: A fault-tolerance approach to reliability of software operation. In *Proc. 8th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-8)*, pages 3–9, 1978. [11](#)
- [14] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riché. UpRight cluster services. In *Proc. of the ACM SOSP’09*, 2009. [1](#)
- [15] Allen Clement, Edmund Wong, Lorenzo Alvisi, Mike Dahlin, and Mirco Marchetti. 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, 2009. [1](#)
- [16] Caio Yuri Silva Costa and Eduardo Adilio Pelinson Alchieri. Diversity on state machine replication. In *The 32nd IEEE International Conference on Advanced Information Networking and Applications*, 2018. [39](#)
- [17] Caio Yuri Silva Costa and Eduardo Adilio Pelinson Alchieri. Architecture for diversity in the implementation of dependable and secure services using the state machine replication approach. *International Journal of Grid and Utility Computing*, Accepted 2019, to appear. [39](#)
- [18] Kevin Driscoll, Brendan Hall, Håkan Sivencrona, and Phil Zumsteg. Byzantine fault tolerance, from theory to reality. In *Computer Safety, Reliability, and Security*, pages 235–248. Springer, 2003. [1](#), [7](#)
- [19] Caio Yuri Silva Costa e Eduardo Adilio Pelinson Alchieri. Implementando diversidade em replicação máquina de estados. In *Anais do XVII Workshop de Testes e Tolerância a Falhas–WTF 2016*, 2016. [39](#)
- [20] Python Software Foundation. ctypes—a foreign function library for python. <https://docs.python.org/2/library/ctypes.html>, 2015. Acessado: 2015-10-18. [15](#)
- [21] Miguel Garcia, Alysson Bessani, Ilir Gashi, Nuno Neves, and Rafael Obelheiro. Os diversity for intrusion tolerance: Myth or reality? In *Proceedings of the IEEE/IFIP 41st International Conference on Dependable Systems&Networks, DSN ’11*, 2011. [1](#), [10](#)

- [22] Miguel Garcia, Alysson Bessani, Ilir Gashi, Nuno Neves, and Rafael Obelheiro. Analysis of operating system diversity for intrusion tolerance. *Software: Practice and Experience*, 44(6):735–770, 2014. 1, 10, 11
- [23] Google. Protocol buffers developers guide. <https://developers.google.com/protocol-buffers/docs/overview>, 2015. Acessado: 2015-06-07. 16, 17
- [24] Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The next 700 BFT protocols. In *Proceedings of the ACM SIGOPS/EuroSys European Systems Conference*, 2010. 1
- [25] Vassos Hadzilacos and Sam Toueg. A modular approach to the specification and implementation of fault-tolerant broadcasts. Technical report, Department of Computer Science, Cornell, May 1994. 7
- [26] Mark Kenneth Joseph. Architectural issues in fault-tolerant, secure computing systems. Technical report, California Univ., Los Angeles, CA (USA), 1988. 11
- [27] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative Byzantine fault tolerance. *ACM Transactions on Computer Systems*, 27(4):7:1–7:39, December 2009. 1
- [28] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982. x, 6
- [29] Jean-Claude Laprie. *Dependability: Basic concepts and terminology*. Springer, 1992. 5
- [30] Siliang Li and Gang Tan. Exception analysis in the java native interface. *Science of Computer Programming*, 89:273–297, 2014. 12
- [31] Rafael Obelheiro, Alysson Neves Bessani, and Lau Cheuk Lung. Analisando a viabilidade da implementação prática de sistemas tolerantes a intrusoes. *Anais do V Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais-SBSeq 2005*, 2005. 1, 11, 12
- [32] Rafael Rodrigues Obelheiro, Alysson Neves Bessani, and Lau Cheuk Lung. 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 - SBSeq 2005*, September 2005. 1, 10, 11
- [33] Oracle. Java native interface specification. <http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html>, 2014. Acessado: 2015-06-01. 14, 15
- [34] Marco Platania, Daniel Obenshain, Thomas Tantillo, Ricky Sharma, and Yair Amir. Towards a practical survivable intrusion tolerant replication system. In *33rd IEEE International Symposium on Reliable Distributed Systems*, pages 242–252, 2014. 1, 10

- [35] Java Native Access project. Jna api documentation—overview. <http://java-native-access.github.io/jna/4.2.1/>, 2015. Acessado: 2016-6-18. 13
- [36] Java Native Access project. Java native access (jna)—readme. <https://github.com/java-native-access/jna/blob/master/README.md>, 2016. Acessado: 2016-6-18. 13
- [37] B. Randell. System structure for software fault tolerance. In *Proceedings of the International Conference on Reliable Software*, 1975. 10
- [38] Fred B. Schneider. Implementing fault-tolerant service using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990. 1, 7, 8
- [39] J. Sousa, A. Bessani, and M. Vukolic. A byzantine fault-tolerant ordering service for the hyperledger fabric blockchain platform. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 51–58, June 2018. 40
- [40] Beth Stearns. Java native interface. <http://www.sc.ehu.es/towmogoj/tutorial%20Java/native1.1/>, 2004. Acessado: 2015-04-09. 13, 14
- [41] Kurt Vanmechelen and Jan Broeckhove. Interfacing c++ member functions with c libraries. *Jack Dongarra University of Tennessee and Oak Ridge National Laboratory and Kaj Madsen and Jerzy Wasniewski*, page 204, 2004. 25
- [42] Kenton Varda. Protocol buffers: Google’s data interchange format. <http://google-opensource.blogspot.com/2008/07/protocol-buffers-googles-data.html>, 2008. Acessado: 2015-04-07. 15
- [43] Giuliana Veronese, Miguel Correia, Alysso Bessani, Lau Lung, and Paulo Verissimo. Efficient Byzantine fault tolerance. *IEEE Transactions on Computers*, 62(1), January 2013. 1
- [44] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proc. of 5th Symp. on Operating Systems Design and Implementations*, 2002. 31, 34, 36

Apêndice A

Demonstração de funcionamento

Neste apêndice será demonstrada a implementação e o funcionamento de uma aplicação com diversidade, utilizando as linguagens suportadas pela solução proposta neste trabalho (C, C++, Java e Python). Para tal, foi desenvolvida uma aplicação de lista com diversidade. Esta aplicação foi implementada nos servidores através de uma lista encadeada. Por brevidade, será mostrada a implementação de apenas uma das operações da lista (ADD). Esta aplicação é a mesma utilizada nos experimentos do Capítulo 4.

A.1 Algoritmo da lista encadeada

A lista foi utilizada para armazenar inteiros e cinco operações foram implementadas para seu acesso: ADD, REMOVE, GET, SIZE e CONTAINS. A lista é representada por uma *struct* (ou classe, no caso de Java e Python) com dois ponteiros/referências: um para o início e outro para o fim da lista. A lista é simplesmente encadeada, com cada elemento contendo um ponteiro para o próximo.

A seguir, o funcionamento de cada operação será explicado detalhadamente.

- CONTAINS: Realiza uma pesquisa linear na lista, percorrendo-a até o fim ou até encontrar o elemento procurado. Retorna 1 caso o inteiro passado como parâmetro exista na lista, e retorna 0 em caso contrário.
- ADD: Inicialmente realiza uma pesquisa linear na lista, para verificar se o elemento a ser adicionado já não existe na lista. Caso não exista, o mesmo é incluído no fim da lista. Retorna 1 caso o elemento tenha sido incluído na lista, ou retorna 0 se o mesmo já existe.
- REMOVE: Realiza uma pesquisa linear na lista, procurando o elemento a ser removido. Retorna 1 caso o elemento tenha sido encontrado e removido, ou retorna 0 caso o elemento não exista.

- GET: Retorna o elemento da posição passada como parâmetro ou um código de erro caso não exista um elemento na posição indicada (posição solicitada maior que o tamanho da lista). Para isso, é realizada uma pesquisa linear com um contador, incrementado a cada elemento percorrido.
- SIZE: Retorna o tamanho da lista. Para isto, a lista é percorrida com um contador, incrementado a cada elemento percorrido. Quando o fim da lista é atingido, o valor do contador é retornado.

A.2 Protocol Buffers

Foram criados três mensagens diferentes, **Estado** (Algoritmo 9), **Request** (Algoritmo 10) e **Response** (Algoritmo 11), representando o Estado da réplica, as requisições e respostas, respectivamente. A palavra-chave `package` define um espaço de nome a ser usado para a classe criada (e.g. em Java, um *package* e em C++, um *namespace*).

Após criados os arquivos `.proto` do Protocol Buffers, os mesmos devem ser compilados para cada uma das linguagens alvo, conforme discutido na Seção 3.3.3. Os arquivos de código-fonte gerados são então utilizados na implementação dos clientes e servidores.

Algoritmo 9 Definição Protocol Buffers para a mensagem Estado

```

1 package bftbench;
2 message Estado
3 {
4     repeated int32 lista = 1;
5 }

```

Algoritmo 10 Definição Protocol Buffers para a mensagem Request

```

1 package bftbench;
2 message Request
3 {
4     enum RequestType {
5         ADD = 0;
6         REMOVE = 1;
7         SIZE = 2;
8         CONTAINS = 3;
9         GET = 4;
10    }
11    required RequestType action = 1;
12    optional int32 value = 2;
13 }

```

Algoritmo 11 Definição Protocol Buffers para a mensagem `Response`

```
1 package bftbench;
2 message Response
3 {
4     optional bool BoolResponse = 1;
5     optional int32 IntResponse = 2;
6 }
```

A.3 Cliente

Os clientes foram implementados para funcionar de forma interativa, *i.e.* solicitando uma entrada do usuário e apresentando o resultado na tela, com exceção do cliente Java, que foi desenvolvido para realizar os *benchmarks* do Capítulo 4.

O cliente em Java ficou responsável pela coleta de dados dos *benchmarks*, e foi baseado em clientes já existentes como parte do projeto BFT-SMART. Mais detalhes sobre seu funcionamento estão disponíveis no Capítulo 4.

De modo a testar o funcionamento da interface com as linguagens C e C++, foi criado um cliente em C. Esse cliente está dividido em duas partes: inicialização (Algoritmo 12), onde é configurado e carregado o BFT-SMART, e execução (Algoritmo 13), onde são feitas as requisições aos servidores. Não foi necessário construir um cliente em C++, pois a interface utilizada em ambos é a mesma.

Por fim foi criado um cliente com a linguagem Python (Algoritmo 14), com a mesma funcionalidade que o cliente em C, solicitando entrada do usuário para executar as operações na lista interativamente.

A.4 Servidor

Na implementação de lista nos servidores, houve o cuidado de utilizar o mesmo algoritmo de lista em todas as linguagens (Algoritmos 15 e 16), de modo a garantir que a complexidade das operações fosse a mesma e não distorcesse os valores aferidos nos experimentos. Desta forma, apesar de em linguagens como Java, C++ e Python existirem estruturas específicas da linguagem de programação que realizem todas as operações, foi implementada a lista encadeada utilizando *structs* e ponteiros em C e C++, e classes e referências em Python e Java.

O servidor em C é dividido em três partes: inicialização (Algoritmo 17), onde é carregado o BFT-SMART, execução (Algoritmo 18), onde são processadas as requisições, e transferência de estado (Algoritmo 19), onde são processadas a criação e instalação

Algoritmo 12 Código de um cliente em C (inicialização do BFT-SMART)

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <bftsmart-wrapper.h>
4 #include <Request.pb-c.h>
5 #include <Response.pb-c.h>
6 #include <Estado.pb-c.h>
7 int main(int argc, char* argv[]) {
8 // argv[1]: id da réplica, argv[2]: classpath para o Java
9     setClasspath(argv[2]);
10    loadJVM();
11    if (createServiceProxy(atoi(argv[1])) != 0) {
12        printf("%s", "Erro ao criar service proxy.\n"); return -2;
13    }
14    BFT_BYTE saida[10000]; // o retorno da chamada é escrito aqui
15    int cmd,result;
16    // lista as operações disponíveis para o usuário
17    do {
18        printf("%s\n", "1-add");
19        printf("%s\n", "2-get");
20        printf("%s\n", "3-remove");
21        printf("%s\n", "4-contains");
22        printf("%s\n", "5-size");
23        printf("%s\n", "6-FINALIZAR. pressione ctrl-c depois.");
```

de estado enviado pelo BFT-SMART durante processo de verificação e recuperação de réplicas.

A implementação do servidor em Python também possui as mesmas partes: inicialização (Algoritmo 20), execução e transferência de estado (Algoritmo 21), porém bem mais enxutas, devido às funcionalidades da linguagem (dinâmica e orientada a objeto) e parte da inicialização ter sido integrada à classe BFTJVM (Algoritmo 22).

A.5 Execução

Para a execução da aplicação, é necessário primeiro iniciar as 4 réplicas (servidores). Após iniciadas as réplicas, são iniciados clientes e realizados testes para verificar se a execução é bem sucedida e se os valores retornados pelas operações são coerentes. Foram realizados testes tanto em ambientes sem diversidade, com todos os servidores na mesma linguagem, e em ambiente com diversidade, com cada servidor em uma linguagem diferente. Para validar o funcionamento dos protocolos de transferência de estado, cada uma das réplicas era reiniciada, alternadamente, após a realização de um conjunto de operações na lista.

Na invocação da biblioteca desenvolvida, é papel do usuário fornecer o caminho para a *classpath* onde as classes do BFT-SMART estarão armazenadas. Além disso, na biblioteca

Algoritmo 13 Código de um cliente em C (somente operação ADD)

```
1 // continua do Algoritmo 12
2 // lê a opção do usuário
3 scanf("%d",&cmd);
4 getchar();
5 // variáveis para as mensagens do Protocol Buffers
6 Bftbench_Request req ; Bftbench_Response *rsp ;
7 unsigned int tamanho ;
8 BFT_BYTE * out ;
9 switch (cmd) {
10 case 1:
11     printf("%s\n", "Digite o valor:"); scanf("%d", &cmd); getchar();
12     // cria a mensagem de requisição
13     bftbench_request_init(&req);
14     req.value = cmd;
15     req.has_value= 1;
16     req.action = BFTBENCH_REQUEST_REQUEST_TYPE_ADD;
17     // converte a mensagem para vetor de bytes.
18     tamanho = bftbench_request_get_packed_size(&req);
19     out = (BFT_BYTE*) malloc (tamanho);
20     bftbench_request_pack(&req, (uint8_t*) out);
21     // executa a operação no servidor
22     result = invokeOrdered(out, tamanho, saida);
23     // le a resposta da operacao e imprime na tela
24     rsp = bftbench_response_unpack(NULL, result, (const uint8_t*)saida);
25     printf("%s%d\n","response:",rsp->boolresponse);
26     // libera a memória alocada
27     bftbench_response_free_unpacked(rsp, NULL);
28     free(out);
29     break;
30 case 2: // ...
31 case 3: // ...
32 case 4: // ...
33 case 5: // ...
34 }
35 } while (cmd != 6);
36 finalizeJVM();
37 return 0;
38 }
```

Python, é necessário fornecer o caminho para a biblioteca compartilhada *libbftsmr.so*, já que por ser uma linguagem interpretada, o carregamento da biblioteca é feito em tempo de execução.

A.6 Considerações

Com a implementação da aplicação de lista, apesar de simples, foi possível validar que a implementação funciona corretamente. Como o sistema continuou operando normalmente e

Algoritmo 14 Código de um cliente em Python (somente operação ADD)

```
1 #!/usr/bin/python
2 from ctypes import *
3 from bftsmart_clientlib import *
4 import Estado_pb2
5 import Request_pb2
6 import Response_pb2
7 # codigo main do programa
8 bc = BFTSMaRtClient(sys.argv[1], sys.argv[2], sys.argv[3])
9 # argv[1]: classpath para o java. argv[2]: id da réplica. argv[3]: caminho da biblioteca de diversidade
10 # apresenta as opções para o usuário
11 mydata = raw_input('acao(add,remove,size,contains,get,sair) :')
12 while mydata != 'sair':
13     if mydata == 'add':
14         # cria a mensagem, preenche com um numero digitado pelo usuário
15         req = Request_pb2.Request()
16         req.action = Request_pb2.Request.ADD
17         req.value = int(raw_input('numero:'))
18         # executa operação no servidor
19         rsp = bc.invokeOrdered(req.SerializeToString())
20         # lê a resposta e imprime na tela
21         rspP = Response_pb2.Response()
22         rspP.ParseFromString(rsp)
23         print 'resposta:'
24         print rspP.BoolResponse
25     if mydata == 'remove':
26         # ...
27     if mydata == 'size':
28         # ...
29     if mydata == 'contains':
30         # ...
31     if mydata == 'get':
32         # ...
33     mydata = raw_input('acao(add,remove,size,contains,get,sair) :')
34 print 'pressione ctrl -c.'
35 bc.finalizeJVM()
```

retornando resultados corretos, ficou evidenciado que o estado era transferido corretamente entre as réplicas.

Algoritmo 15 Implementação de lista em C (somente operação ADD)

```
1 typedef struct elemento {
2     int dado;
3     struct elemento * proximo;
4 } t_elemento;
5 typedef struct lista {
6     t_elemento * inicio ;
7     t_elemento * fim;
8 } t_lista ;
9 int insereFinal(int valor, t_lista * l) {
10     t_elemento * novoultimo = (t_elemento *)malloc(sizeof(t_elemento));
11     novoultimo->dado = valor;
12     novoultimo->proximo = NULL;
13     if(l->inicio == NULL)
14         l->inicio = novoultimo;
15     else
16         l->fim->proximo = novoultimo;
17     l->fim = novoultimo;
18     return 0;
19 }
20 int buscarIndice(int valor, t_lista * l) {
21     t_elemento * atual = l->inicio;
22     int i = 0;
23     while (atual != NULL) {
24         if (atual->dado == valor) return i;
25         i++;
26         atual = atual->proximo;
27     }
28     return -1;
29 }
```

Algoritmo 16 Implementação da lista em Python (somente operação ADD)

```
1 class Elemento(object):
2     valor = 0
3     proximo = None;
4 class ListaNormal(object):
5     inicio = None
6     fim = None
7     def insereFinal(self , valor):
8         novoutimo = Elemento()
9         novoutimo.valor = valor
10        novoutimo.proximo = None
11        if self.inicio is None:
12            self.inicio = novoutimo
13        else:
14            self.fim.proximo = novoutimo
15            self.fim = novoutimo
16        return 0
17    def buscarIndice(self , valor):
18        atual = self.inicio
19        i = 0
20        while atual is not None:
21            if atual.valor == valor:
22                return i
23            i += 1
24            atual = atual.proximo
25        return -1
```

Algoritmo 17 Servidor em C (inicialização)

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <bftsmart-wrapper.h>
4 #include <Request.pb-c.h>
5 #include <Response.pb-c.h>
6 #include <Estado.pb-c.h>
7 // omitida a implementação de funções do servidor (Algoritmos 18 e 19).
8 int main(int argc, char* argv[]) {
9     // argv[1]: id da replica, argv[2]: classpath para o Java
10    // cria a estrutura de lista, usada no Algoritmo 18
11    state = criarLista();
12    // inicia a JVM
13    setClasspath(argv[2]);
14    loadJVM();
15    // registra as funções do servidor junto ao bft-smart
16    implementExecuteOrdered(&execute);
17    implementExecuteUnordered(&execute);
18    implementInstallSnapshot(&installSnap);
19    implementgetSnapshot(&getSnap);
20    // inicia a réplica
21    startServiceReplica(atoi(argv[1]));
22    finalizeJVM();
23    return 0;
24 }
```

Algoritmo 18 Servidor em C (somente operação ADD).

```
1 // .... omitida a implementação da lista (Algoritmo 15)
2 t_lista * state;
3 int execute(BFT_BYTE cmd[], int siz, BFT_BYTE ** mem) {
4     // lê a mensagem de requisição, convertendo de vetor de bytes para Protocol Buffers, determina a
5     // ação a ser executada, e a executa.
6     Bftbench_Request *msg = bftbench_request_unpack(NULL, siz, (const uint8_t*)cmd);
7     Bftbench_Response rsp = BFTBENCH_RESPONSE_INIT;
8     int x;
9     switch (msg->action)
10    {
11        // executa a inserção na lista e retorna se foi bem-sucedido
12        // trabalha diretamente com os objetos Protocol Buffers
13        case BFTBENCH_REQUEST_REQUEST_TYPE_ADD:
14            x = buscarIndice(msg->value, state);
15            rsp.has_boolresponse = 1;
16            rsp.boolresponse = 0;
17            if (x < 0) {
18                rsp.boolresponse = 1;
19                insereFinal(msg->value, state);
20            }
21            break;
22        // case REMOVE, case CONTAINS, case SIZE e case GET omitidos.
23    }
24    // converte a resposta em vetor de bytes, e retorna para a função chamadora
25    unsigned int tamanho = bftbench_response_get_packed_size(&rsp);
26    BFT_BYTE * out = (BFT_BYTE*) malloc (tamanho);
27    bftbench_response_pack(&rsp, (uint8_t*) out);
28    (*mem) = out;
29    return tamanho;
30 }
```

Algoritmo 19 Servidor em C (funções de transferência de estado)

```
1 void installSnap(BFT_BYTE stateNovo[], int siz) {
2     // abre a mensagem Protocol Buffers
3     Bftbench_Estado *msg = bftbench_estado_unpack(NULL, siz, (const uint8_t*)stateNovo);
4     // destrói e recria a lista com os valores recebidos
5     destruirLista (state);
6     state = criarLista();
7     unsigned int i;
8     for ( i = 0; i < msg->n_lista; i++) {
9         insereFinal(msg->lista[i], state);
10    }
11    // libera a memória alocada
12    bftbench_estado_free_unpacked(msg, NULL);
13 }
14 int getSnap(BFT_BYTE ** mem) {
15     // cria uma mensagem vazia Protocol Buffers e aloca espaço para a lista
16     Bftbench_Estado est = BFTBENCH_ESTADO_INIT;
17     int x = tamanhoLista(state);
18     est.n_lista = x;
19     est.lista = (int32_t*) malloc(sizeof(int) * est.n_lista);
20     // copia todos os elementos da lista para a mensagem Protocol Buffers
21     int i = 0;
22     t_elemento * atual = state->inicio;
23     while (atual != NULL) {
24         est.lista [i] = atual->dado;
25         i++;
26         atual = atual->proximo;
27     }
28     // converte a mensagem para vetor de bytes e retorna ao bft-smart.
29     unsigned int tamanho = bftbench_estado_get_packed_size(&est);
30     BFT_BYTE * out = (BFT_BYTE*) malloc (tamanho);
31     bftbench_estado_pack(&est, (uint8_t*)out);
32     free (est.lista);
33     (*mem) = out;
34     return tamanho;
35 }
```

Algoritmo 20 Script principal (*main*) do servidor em Python.

```
1 import sys
2 from list_imp import *
3 # argv[1]: classpath do Java. argv[2]: id da réplica. argv[3]: caminho da biblioteca de diversidade.
4 # inicia o servidor.
5 bc = BFTList(sys.argv[1], sys.argv [2], sys.argv [3])
6 bc.finalizeJVM()
```

Algoritmo 21 Implementação do servidor em Python (somente operação ADD)

```
1 import sys
2 import Estado_pb2
3 import Request_pb2
4 import Response_pb2
5 from bftsmart_serverlib import *
6 class BFTList(BFTSMaRtServer):
7     def __init__(self, clspath, id, dllpath):
8         # chama o construtor da classe base
9         super(BFTList, self).__init__(clspath, id, dllpath)
10        self.state = ListaNormal()
11    def execute(self, input):
12        req = Request_pb2.Request()
13        req.ParseFromString(input)
14        if req.action == Request_pb2.Request.ADD:
15            val = req.value;
16            x = self.state.buscarIndice(val)
17            rsp = Response_pb2.Response()
18            rsp.BoolResponse = False
19            if x < 0:
20                rsp.BoolResponse = True
21                self.state.inserFinal(val)
22            return rsp.SerializeToString()
23    def invokeOrdered(self, input):
24        return self.execute(input)
25    def invokeUnordered(self, input):
26        return self.execute(input)
27    # lê as informações da lista e retorna para o bft-smart.
28    def getSnapshot(self):
29        est = Estado_pb2.Estado()
30        atual = self.state.inicio
31        while atual is not None:
32            est.lista.append(atual.valor)
33            atual = atual.proximo
34        return est.SerializeToString()
35    # lê as informações recebidas e re-cria a lista com elas.
36    def installSnapshot(self, input):
37        est = Estado_pb2.Estado()
38        est.ParseFromString(input)
39        self.state = ListaNormal()
40        for i in est.lista:
41            self.state.inserFinal(i)
```

Algoritmo 22 Código da classe BFTJVM (simplificado—algumas funções omitidas)

```
1 class BFTJVM(object):
2     # parametros: dllpath: caminho para a biblioteca de diversidade
3     # classpth : classpath para inicializar a JVM.
4     def __init__(self, dllpath, classpth):
5         # carrega a biblioteca de diversidade
6         BFTJVM.libbft = CDLL(dllpath);
7         # configura o tipo de retorno de cada função em C.
8         # None equivale a void, e c_void_p equivale a (void *)
9         BFTJVM.libbft.setClasspath.restype = None
10        BFTJVM.libbft.bftsmartallocate.restype = c_void_p
11        BFTJVM.libbft.bftsmartrelease.restype = None
12        BFTJVM.libbft.finalizeJVM.restype = None
13        BFTJVM.libbft.implementExecuteOrdered.restype = None
14        # executa a função setClasspath, convertendo string Python para string C (char*)
15        BFTJVM.libbft.setClasspath(c_char_p(classpth));
16        # carrega a JVM
17        BFTJVM.libbft.loadJVM()
18 # define tipos de callback, que serão utilizados para criar ponteiros
19 # para função que permitem ao C executar as funções em Python
20 INVOKORDFUNC = CFUNCTYPE(c_int, POINTER(c_char), c_int, POINTER(POINTER(c_char)))
21 INVOKUNORDFUNC = CFUNCTYPE(c_int, POINTER(c_char), c_int,
22     POINTER(POINTER(c_char)))
23 GETSNAPFUNC = CFUNCTYPE(c_int, POINTER(POINTER(c_char)))
```
