Confiabilidade de Sistemas Distribuídos

Ano Letivo: 2017/2018

Trabalho Prático nº 2

Privacy-Enhanced D-SKVS

A Dependable Searchable Homomorphic Key Value Store

Grupo: G01

Nº 42009 – João Almeida
Nº 41803 – Jorge Valadas
Nº 41959 – Mário Carvalho

Resumo

O trabalho realizado para a segunda fase consiste na adição de mecanismos de privacidade ao data storage implementado para a primeira fase. Estes mecanismos consistem na conversão das operações realizadas pelo cliente em operações sobre dados cifrados. No final é apresentada uma avaliação experimental comparando as duas implementações em situações diferentes.

1. Introdução e contexto do trabalho

O principal objetivo desta fase foi a introdução de mecanismos adicionais de privacidade sobre os dados guardados na Key-Value Storage implementada na primeira fase. Foi introduzido um proxy de forma a efetuar a transformação das operações introduzidas pelo cliente de forma a que fossem suportadas as mesmas operações, mas utilizando as funções disponibilizadas em [2]. Foi desenvolvida uma interface Rest para o proxy para que o código do lado do cliente permanecesse igual. Ao receber um pedido, o proxy irá comunicar a operação convertida ao servidor e o processamento destas operações é feito de uma forma semelhante ao da primeira fase, à exceção de algumas operações, como é o caso da soma em que o servidor fazia um pedido GET para as réplicas e somava os valores e nesta fase a réplica irá efetuar a operação de soma. Para que os dados sejam cifrados é necessária a existência de chaves. As chaves destes métodos foram inicializadas no Proxy e os

parâmetros necessários para efetuar as operações foram também inicializados nas réplicas.

2. Modelo e arquitetura do sistema

Como apresentado na Figura 1 o sistema é composto 4 componentes principais o Cliente, que irá efetuar os pedidos, o Proxy que transforma as operações para suportar criptografia homomórfica parcial, o Servidor que recebe os pedidos e faz as invocações nas Réplicas e as Réplicas que armazenam os dados utilizando o Redis como armazenamento e efetuam operações sobre os dados cifrados.

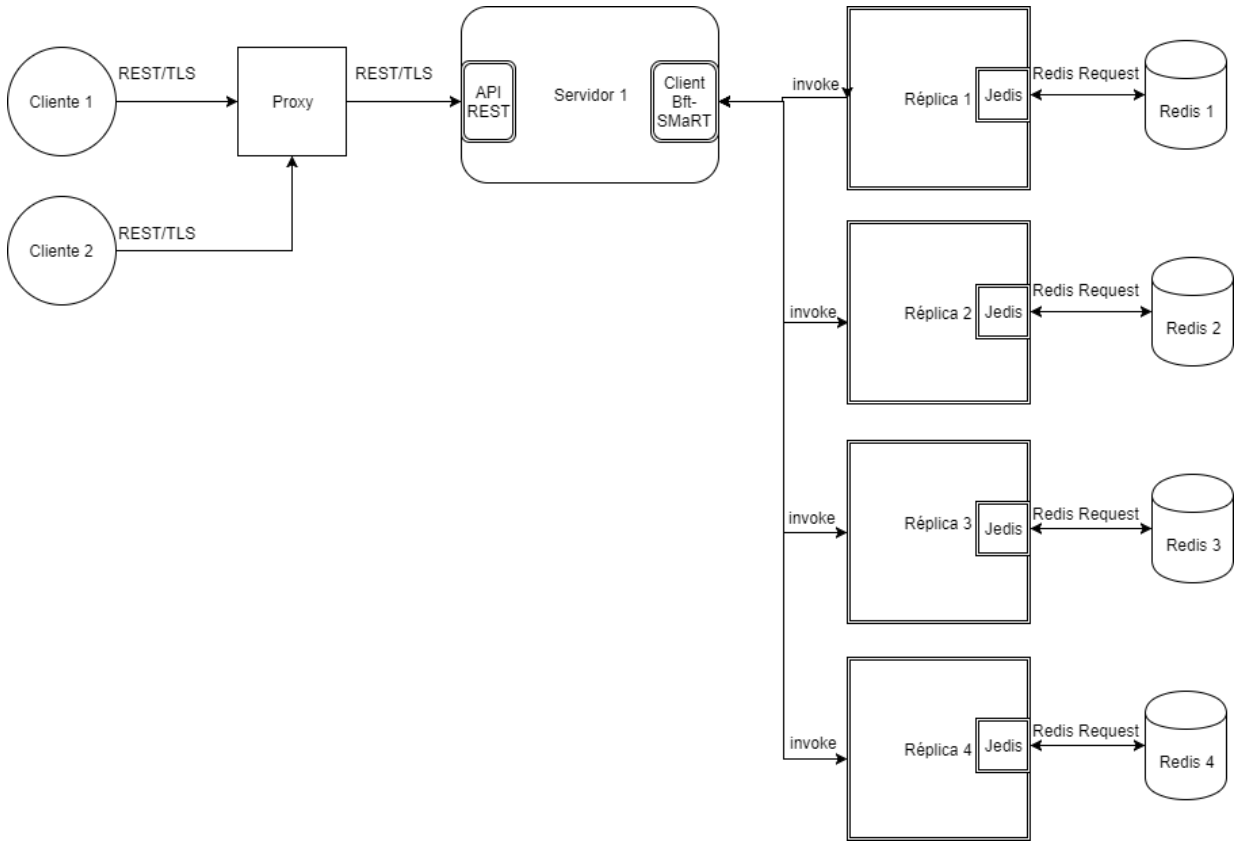

Figure 1 Arquitetura do Sistema com 2 clientes e 4 réplicas

3. APIs para integração dos clientes

Nesta secção encontra-se a interface Rest que se encontra no Proxy e que permite a um cliente efetuar pedidos de forma a utilizar as operações implementadas.

```
@POST
@Path("/ps")
@Consumes(MediaType.APPLICATION_JSON)
public void putSet( Entry entry)
@GET
@Path("/{id}")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public byte[] getSet(@PathParam("id") String id)

@POST
@Path("/adde/{id}")
@Consumes(MediaType.APPLICATION_JSON)
public Response addElement( @PathParam("id") String id)

@GET
@Path("/{id}/{pos}")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public String readElement(@PathParam("id") String id, @PathParam("pos") int
pos)

@GET
@Path("/ie/{id}/{element}")
@Consumes(MediaType.APPLICATION_JSON)
public String isElement(@PathParam("id") String id, @PathParam("element")
String element)

@PUT
@Path("/{id}/{pos}")
@Consumes(MediaType.APPLICATION_JSON)
public Response writeElement(@PathParam("id") String id, @PathParam("pos")
int pos, String new_element)

@DELETE
@Path("/rs/{id}")
@Consumes(MediaType.APPLICATION_JSON)
public Response removeSet(@PathParam("id") String id)

@GET
@Path("/sum/{id1}/{id2}/{pos}")
@Produces(MediaType.APPLICATION_JSON)
public byte[] sum(@PathParam("id1") String id1, @PathParam("id2") String
id2, @PathParam("pos") int pos)

@GET
@Path("/mult/{id1}/{id2}/{pos}")
@Produces(MediaType.APPLICATION_JSON)
public byte[] mult(@PathParam("id1") String id1, @PathParam("id2") String
id2, @PathParam("pos") int pos)
```

```
@GET
@Path("/seq/{pos}/{val}")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public String searchEq(@PathParam("pos") int pos, @PathParam("val") String
val)

@GET
@Path("/sbt/{pos}/{val}")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public String searchBt(@PathParam("pos") int pos, @PathParam("val") String
val)

@GET
@Path("/slt/{pos}/{val}")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public String searchLt(@PathParam("pos") int pos, @PathParam("val") String
val)
```

4. Modelo de falhas e modelo de adversário (ou modelo de confiabilidade)

Modelo de falhas: o objetivo é o sistema suportar falhas por crash e falhas bizantinas para $n \geq 3f + 1$, onde n representa o número de servidores e f representa o número de replicas *faulty*. No entanto, não foi possível suportar falhas por crash. Este suporte poderia ter sido implementado alterando as configurações do Bft-SMaRt ao alterar o tempo de espera por uma resposta da réplica até que esta réplica fosse removida e inserida uma nova.

Modelo de adversário: o modelo de adversário para este projeto são ataques às comunicações entre o cliente e o proxy, o proxy e o servidor e entre réplicas. Também foi considerado um administrador honesto mas curioso com acesso aos valores armazenados nas réplicas.

5. Detalhe da arquitetura do sistema e seus componentes

Existem 4 componentes principais no sistema: o cliente, o proxy, o servidor e as réplicas. O cliente consiste na entidade que comunica com o proxy através da interface Rest e que irá efetuar operações sobre os dados armazenados. O proxy é o componente que recebe as operações efetuadas pelo cliente e que efetua transformações sobre os dados de forma a cifrá-los para que estes possam suportar operações efetuadas sobre eles. O proxy também possui as

chaves necessárias para realizar as operações de cifra e decifra. De seguida irá comunicar com o servidor através de pedidos Rest. O servidor irá receber os pedidos efetuados e efetuar as respetivas operações nas réplicas. As réplicas irão realizar as operações recebidas sobre os dados cifrados. Cada réplica possui parâmetros das chaves utilizadas para cifrar e decifrar no proxy de forma a que seja possível operar sobre os dados cifrados que se encontram armazenados. Os dados encontram-se armazenados, à semelhança do TP1, num servidor Redis do tipo <String, String> e cada réplica terá um servidor Redis associado. As réplicas, conseguem chegar a consenso através da biblioteca Bft-SMaRt, integrada na primeira fase.

5.1 Implementação do suporte para operações sobre dados cifrados

O suporte para operações de criptografia homomórfica parcial foi desenvolvido tendo como base a biblioteca descrita em [2]. São suportados 5 tipos de encriptação: Search, Sum, Mult, Order Preservation e Random. De forma a suportar operações de pesquisa, existe um campo encriptado utilizando o método disponibilizado em HomoSearch, que irá encriptar o campo e suportar operações de pesquisa de um dado valor. A operação de soma, utiliza como recurso a implementação HomoAdd, utilizando uma chave de Paillier para cifrar e decifrar. A operação de soma é efetuada na réplica utilizando apenas um parâmetro dessa chave, o NSquare. A operação de multiplicação utiliza a implementação em HomoMult e a chave pública para encriptar no Proxy e efetuar a operação de multiplicação na réplica e a chave privada para desencriptar o resultado. As operações de Order Preservation como Search Bigger Than e Search Smaller Than, são cifradas e decifradas no Proxy e a operação de comparação ocorre na réplica. Nestas últimas operações são os valores transferidos para as réplicas são também serializados utilizando a biblioteca HomoSerial. Os campos que são inseridos posteriormente são cifrados e decifrados utilizando a cifra Random de HomoRand que utiliza uma chave e um vetor de inicialização.

5.2 Implementação do suporte para disponibilidade permanente com adição dinâmica de réplicas

Esta funcionalidade não foi implementada, no entanto, a forma correta de a implementar seria efetuar uma modificação ao código da

biblioteca do Bft-SMaRt [1] de forma a que possam ser considerados servidores definidos no ficheiro de configuração que não participam, inicialmente no processo de consenso. Estes servidores sentinent iriam ser inicializados com o sistema, mas aguardariam por um sinal. Existiria uma entidade que receberia os valores calculados pelas réplicas e quando existisse uma réplica que envia valores incorretos esta seria substituída pela réplica sentinent.

6. Teste e avaliação experimental

O ambiente de teste usado foi apenas uma máquina, no entanto, foi utilizado o Docker de forma a simular, exceto nas comunicações efetuadas um ambiente distríbuido. Foram utilizadas 7 réplicas, 1 proxy e 2 clientes de teste. Cada cliente efetuou cada benchmark 3 vezes e foi utilizado o valor médio de throughput em operações por segundo para a avaliação experimental. Foram realizados 3 testes: Sem falhas, 1 servidor bizantino e 2 servidores bizantinos.

Os benchmarks efetuados foram:

- Benchmark 1: 100 PutSet.
- Benchmark 2: 100 GetSet.
- Benchmark 3: 50 PutSet, 50 GetSet alternados.
- Benchmark 4: 100 operações alternadas de: PutSet, GetSet, IsElement, ReadElement, WriteElement, AddElement e RemoveSet.
- Benchmark 5: Benchmark 4 + 10% operações de Mult.
- Benchmark 6: Benchmark 4 + 10% operações de Sum.
- Benchmark 7: 100 operações de SearchEq, SearchBt e SearchLt.
- Benchmark 8: 100 operações alternadas de: PutSet, GetSet, IsElement, ReadElement, WriteElement, AddElement, SearchEq, SearchBt e SearchLt + 10% operações de Sum + 10% operações de RemoveSet.
- Benchmark 9: 100 operações de SearchBt e SearchLt.

6.1 Teste sem falhas de réplicas

Na Figura 2 é possível observar a comparação de throughput em operações/segundo entre a primeira e a segunda fase quando não existem falhas nas réplicas. Existe um overhead causado pela utilização das operações cifradas, desta forma, o número de operações realizadas por segundo são

menores na segunda fase. A complexidade das operações aumentou na segunda fase, pois existem operações que envolvem um maior número de comunicações como é o caso do WriteSet onde é efetuado um pedido Get para obter a linha correspondente, o campo é desencriptado, alterando-o para o novo valor e novamente encriptado e é efetuado um pedido Put. Outro fator que diminui o throughput da segunda fase é o fato das operações, ao operarem sobre dados cifrados, acarretarem uma maior complexidade.

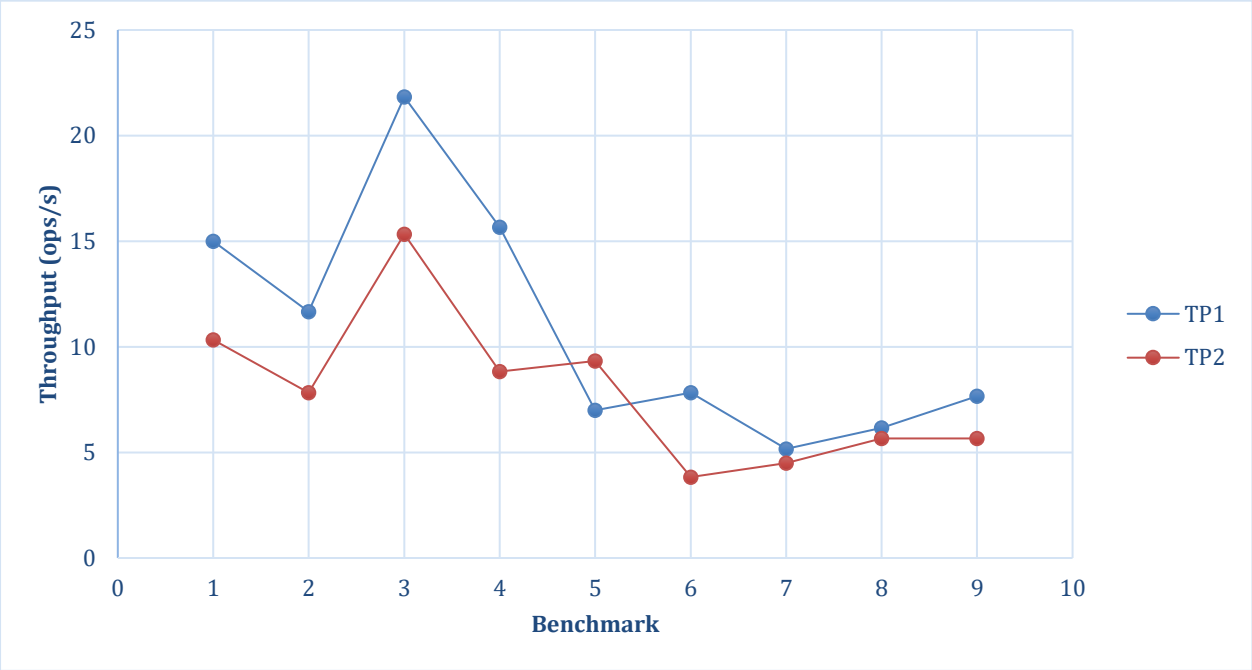

Figure 2 Teste sem falhas.

6.2 Teste com 1 réplica bizantina

Neste teste torna-se mais notória a discrepância entre os dois resultados, dada a inserção de uma réplica bizantina. Dado que uma réplica se encontra a enviar valores errados, existe uma maior complexidade no protocolo de consenso.

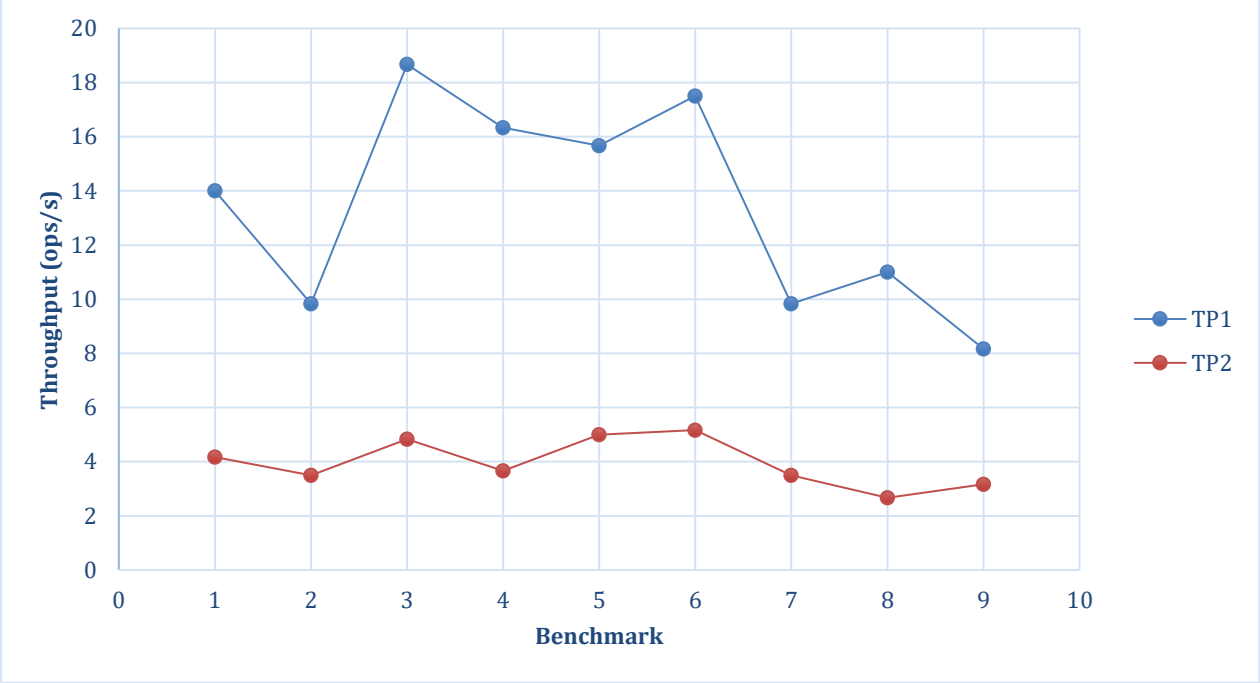

Figure 3 Teste c/ 1 falha bizantina.

6.3 Teste com 2 réplicas bizantinas

Neste teste, onde foram inseridas duas réplicas bizantinas, os resultados são semelhantes aos do teste 6.2.

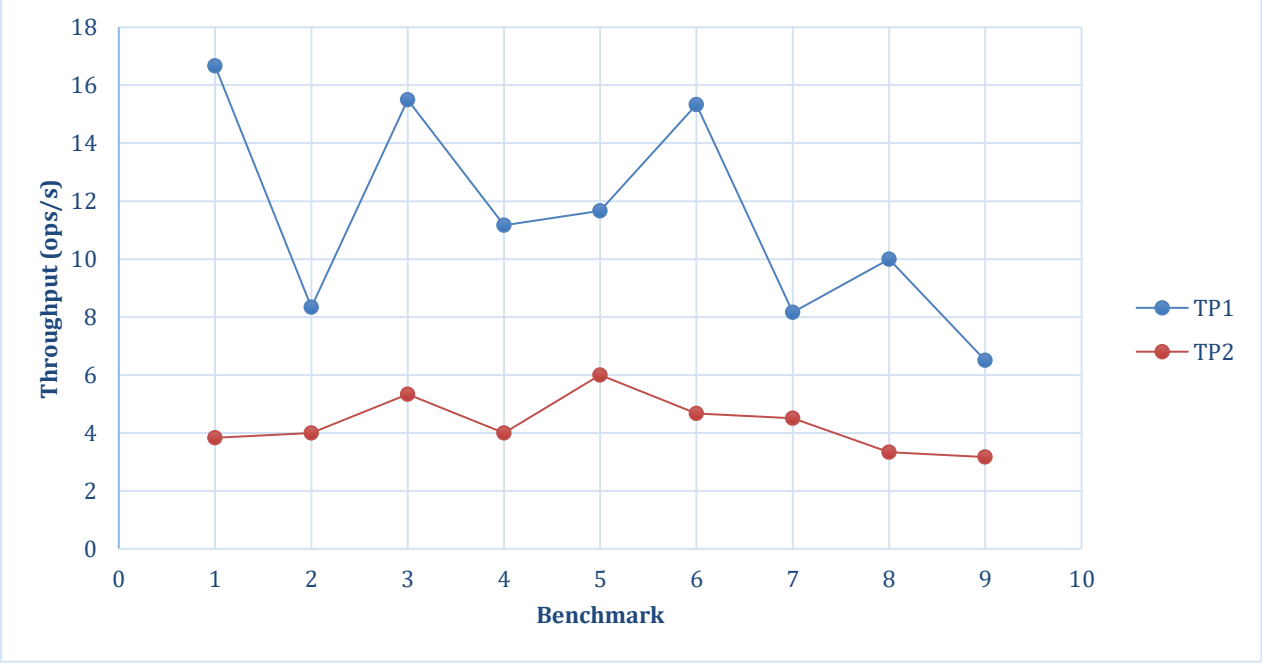

Figure 4 Teste c/ 2 falhas bizantinas.

7. Conclusões

Este projeto foi de extrema importância como forma de perceber o funcionamento de um sistema tolerante a falhas bizantinas. A integração da biblioteca Bft-SMaRt permitiu a simplificação da implementação e a configuração das propriedades do sistema.
A avaliação experimental foi um bom exercício, pois tornou óbvio o overhead causado pelas operações homomórficas parciais.
Apesar das funcionalidades não terem sido integralmente implementadas, os conceitos foram apreendidos e poderão ser postos em prática numa implementação futura.

Referências

[1] A. Bessani, J. Sousa, E. Alchieri, State Machine Replication for the Masses with BFT-SMART , in Proceedings of DSN 2014 , 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, Atlanta, USA, June 2014

[2] SJHomoLib – *Synthetic Java Homomorphic Library*, TR DI-FCT-UNL e respetiva implementação disponibilizada em aula prática (laboratório)