Provendo Segurança e Privacidade em Coordenação Distribuída e Extensível

Edson Floriano S. Junior[1], **Eduardo Alchieri**[1], **Diego F. Aranha**[2], **Priscila Solis**[1]

[1] Departamento de Ciência da Computação – Universidade de Brasília – UnB
[2]Instituto de Computação – Universidade Estadual de Campinas – UNICAMP

Abstract. *Mechanisms for coordination and synchronization, like shared counters and distributed queues, are used in the development of distributed systems. These mechanisms are implemented through coordination infrastructures, such as tuple spaces. A tuple space is a shared memory object that provides operations to store and retrieve ordered sets of data, called tuples. Although tuple spaces provide functionalities for coordination, recent studies have shown that extensible protocols and architectures are fundamental for system performance. The main idea is to allow servers, supporting the coordination infrastructure, to access and process coordination information. Thus, it is not necessary neither to transfer information to clients or to reprocess requests due to concurrent accesses. Existing proposals for extensible distributed coordination do not provide security and privacy once servers must access plain-data. This work proposes the use of robust cryptographic schemes, implemented in* DEPSPACE*, to develop secure protocols for extensible coordination. Experiments show that the proposed solutions significantly improve system performance.*

Resumo. *Mecanismos de coordenação e sincronização, como contadores compartilhados e filas distribuídas, são empregados no desenvolvimento de vários sistemas distribuídos. Estes mecanismos são suportados por infraestruturas de coordenação, como as baseadas em espaço de tuplas. Um espaço de tuplas é um objeto de memória compartilhada que fornece operações para armazenar e recuperar conjuntos ordenados de dados, chamados tuplas. Apesar de espaços de tuplas proverem as funcionalidades necessárias para coordenação, estudos recentes mostraram que o emprego de protocolos e arquiteturas extensíveis é fundamental para o desempenho do sistema. A ideia principal das extensões é permitir que os servidores, que mantém a infraestrutura de coordenação, acessem e processem as informações de coordenação. Desta forma, não é necessário transportar informações para os clientes, além de evitar reprocessamentos devido a acessos concorrentes. As propostas existentes para coordenação distribuída e extensível não fornecem segurança e privacidade adequadas, uma vez que dados em claro são acessados pelos servidores. Neste sentido, este trabalho propõe a utilização de esquemas robustos de criptografia, implementados no* DEPSPACE*, para o desenvolvimento de protocolos de coordenação extensível com propriedades de segurança e privacidade. Experimentos mostram que as soluções propostas melhoram significativamente o desempenho do sistema.*

1. Introdução

Os projetistas e desenvolvedores de aplicações distribuídas estão cada vez mais preocupados com aspectos de segurança e privacidade dos dados manipulados por estas aplicações.

Um sistema é dito seguro se satisfaz requisitos de integridade, disponibilidade e confidencialidade [Avizienis et al. 2004]. Intuitivamente, privacidade é entendida na perspectiva de uma entidade (ex.: pessoas e empresas) como a confidencialidade de suas informações sensíveis (dados e metadados) [Veríssimo 2016]. A existência de muitos dados correlacionados disponíveis, bem como a maior exposição das entidades, são fatores que aumentam o risco à segurança das aplicações [Veríssimo 2016]. Em meio a este cenário, é imperativo que se proteja toda informação disponibilizada para uma aplicação, pois ataques de inferência estatística, através da análise e correlação de dados de acesso público, muitas vezes conseguem obter informações mantidas em segredo [Naveed et al. 2015].

Estes aspectos são particularmente relevantes quando consideramos mecanismos de coordenação e sincronização usados por processos em vários sistemas distribuídos. Estes mecanismos são suportados por infraestruturas de coordenação, como o ZooKeeper [Hunt et al. 2010] e o DEPSPACE [Bessani et al. 2008], e geralmente não fornecem nenhuma garantia de segurança, i.e., os estados dos processos ficam expostos no sistema. Dentre as infraestruturas de coordenação, este trabalho aborda coordenação através de espaços de tuplas, que são objetos de memória compartilhada que fornecem operações para armazenar e recuperar conjuntos ordenados de dados, chamados tuplas. Dentre as propostas existentes para espaços de tuplas, o sistema DEPSPACE [Bessani et al. 2008] será utilizado por prover um maior nível de segurança.

Apesar de espaços de tuplas fornecerem um modelo voltado para a coordenação de processos, estudos recentes mostraram que o emprego de protocolos e arquiteturas extensíveis é fundamental para o desempenho do sistema [Distler et al. 2015]. As extensões possibilitam que os servidores, que mantém a infraestrutura de coordenação, acessem e processem as informações de coordenação. Desta forma, não é necessário transportar informações para os clientes, além de evitar reprocessamentos devido a acessos concorrentes. Extensões foram aplicadas em muitos sistemas e contextos para aumentar tanto o desempenho quanto as funcionalidades dos sistemas [Baldoni et al. 2003, Bershad et al. 1995]. Dentro do contexto de coordenação de processos, extensões foram aplicadas nos sistemas DEPSPACE e ZooKeeper visando aumentar o desempenho de aplicações de coordenação distribuída [Distler et al. 2015]. No entanto, estas propostas existentes para coordenação distribuída e extensível não fornecem segurança e privacidade adequadas, uma vez que dados em claro são acessados pelos servidores.

Neste sentido, este trabalho propõe a utilização de esquemas robustos de criptografia, recentemente integrados ao DEPSPACE [Floriano et al. 2017a, Floriano et al. 2017b], para o desenvolvimento de protocolos de coordenação extensível com propriedades de segurança e privacidade. São apresentados protocolos para contadores compartilhados e filas distribuídas, mecanismos que podem ser usados, por exemplo, em um *data center* para coordenação e sincronização de processos. Experimentos mostram que as soluções propostas melhoram significativamente o desempenho do sistema.

O restante deste trabalho está organizado da seguinte forma. A Seção 2 detalha o conceito de espaço de tuplas. O DEPSPACE, com as recentes extensões para segurança e privacidade, é discutido na Seção 3. A Seção 4 discute coordenação distribuída extensível e apresenta a implementação de dois mecanismos de coordenação (contador compartilhado e fila distribuída). A Seção 5 apresenta os experimentos realizados com as implementações desenvolvidas. Finalmente, as conclusões são apresentadas na Seção 6.

2. Espaço de Tuplas

Um espaço de tuplas [Gelernter 1985] é um objeto de memória compartilhada que fornece operações para armazenar e recuperar conjuntos de dados ordenados chamados de tuplas, permitindo a coordenação de processos de um sistema distribuído desacoplada no espaço (os processos não precisam conhecer as localizações uns dos outros) e no tempo (os processos não precisam estar ativos ao mesmo tempo). Uma tupla t é uma sequência ordenada de campos, onde um campo que contém um valor é dito definido. Um tupla onde todos os campos são definidos é chamada de entrada. Uma tupla $\bar{t}$ é chamada molde (ou *template*) se algum de seus campos não tem valor definido. Diz-se que uma tupla t e um molde $\bar{t}$ combinam se e somente se ambos têm o mesmo número de campos e todos os valores e tipos dos campos definidos em $\bar{t}$ são iguais aos valores e tipos dos campos correspondentes em t. Por exemplo, a tupla $\langle \text{SBRC}, 2018, \text{Campos do Jordão} \rangle$ combina com o molde $\langle \text{SBRC}, 2018, * \rangle$ ('$*$' denota um campo sem definição do molde).

Um espaço de tuplas funciona como uma memória associativa (os dados são acessados a partir de seu conteúdo, e não através de seu endereço), sendo manipulado através das seguintes operações [Gelernter 1985]: $out(t)$ que adiciona a entrada t no espaço de tuplas; $in(\bar{t})$, que remove do espaço de tuplas uma tupla que combina com o molde $\bar{t}$; $rd(\bar{t})$, usada na leitura de uma tupla que combina com o molde $\bar{t}$, sem removê-la do espaço. As operações in e rd são bloqueantes, i.e., se não houver uma tupla que combine com o molde no espaço, o processo fica bloqueado até que uma esteja disponível. Também existem variantes não bloqueantes das operações de leitura, denominadas inp e rdp, que retornam nulo quando não existe uma tupla que combine com o molde. Uma extensão comum a este modelo é a inclusão das seguintes operações [Bessani et al. 2008, Distler et al. 2015, Bakken and Schlichting 1995, Segall 1995]: $cas(\bar{t}, t)$ que insere t no espaço se não tiver uma tupla t' que combine com o molde $\bar{t}$ e retorna nulo, caso contrário retorna t'; $replace(t, t')$ que insere a tupla t' e remove (e retorna) a tupla t caso t esteja no espaço, caso contrário apenas retorna nulo; e $readAll(\bar{t})$ que retorna todas as tuplas que combinem com o molde $\bar{t}$.

3. DEPSPACE: Um Sistema de Coordenação Seguro e Tolerante a Falhas Bizantinas

Segurança é uma característica fundamental de sistemas confiáveis [Avizienis et al. 2004]. No contexto de um espaço de tuplas, os seguintes atributos são necessários: *confiabilidade* (as operações realizadas no espaço de tuplas fazem com que seu estado se modifique de acordo com a especificação), *disponibilidade* (o espaço de tuplas sempre está pronto para executar as operações requisitadas por partes autorizadas), *integridade* (nenhuma alteração imprópria no estado de um espaço de tuplas pode ocorrer), *confidencialidade* (o conteúdo dos campos das tuplas não pode ser revelado a partes não autorizadas). O DEPSPACE [Bessani et al. 2008, Floriano et al. 2017a, Floriano et al. 2017b] consiste na implementação de um espaço de tuplas que busca satisfazer estas propriedades por meio de diversas camadas, cada uma responsável pela concretização de uma funcionalidade ou propriedade de segurança distinta. Estas camadas são descritas a seguir, com uma maior ênfase para a camada responsável pela confidencialidade dos dados armazenados por ser extensivamente usada neste trabalho.

Replicação. Para manter a consistência do espaço de tuplas, o DEPSPACE utiliza replicação Máquina de Estados [Schneider 1990, Castro and Liskov 2002, Bessani et al. 2014]. Este mecanismo está relacionado principalmente com as propriedades de disponibilidade e confiabilidade, pois para um número n de réplicas no sistema, garante que o espaço de tuplas executa as operações a ele endereçadas seguindo sua especificação, mesmo que até $f = (n-1)/3$ réplicas sejam maliciosas (as réplicas corretas *mascaram* o comportamento das maliciosas). Através deste protocolo, as réplicas corretas executam a mesma sequência de operações e retornam os mesmos valores, evoluindo de forma sincronizada.

Confidencialidade. Para evitar pontos únicos de falha, a preservação da propriedade de confidencialidade não é atribuída a um único servidor, mas a um conjunto deles. Sendo assim, a confidencialidade é implementada através do uso de um (n,$f+1$) – esquema de compartilhamento de segredo publicamente verificável (*publicly verifiable secret sharing* – PVSS) [Schoenmakers 1999]. Através deste esquema, os clientes cifram as tuplas com um segredo por eles gerado e geram um conjunto de n fragmentos (*shares*) deste segredo. O segredo pode ser remontado apenas com a combinação de $f+1$ *shares*, o que torna impossível que um conluio de até f servidores faltosos revele o conteúdo de uma tupla. Como os servidores não conseguem acessar o conteúdo da tupla, que está cifrado, o DEPSPACE utiliza um *fingerprint* da tupla para possibilitar a comparação entre tuplas e moldes. Este *fingerprint* é computado de acordo com os seguintes tipos de campos escolhidos para a tupla [Bessani et al. 2008, Floriano et al. 2017a, Floriano et al. 2017b]:

- Público (PU): o próprio valor do campo é o *fingerprint*, i.e, nenhum método criptográfico é aplicado ao conteúdo do campo que fica exposto.
- Comparável (CO): um *hash* do valor do campo é o *fingerprint* (para isso utiliza uma função de *hash* resistente a colisões), possibilitando a execução de buscas (comparações).
- Comparável Determinístico (CD): o conteúdo do campo é cifrado, através da função $encryptCD(key_{shared}, content)$, com um algoritmo determinístico de criptografia simétrica [Boneh and Shoup 2015] e o resultado é usado como *fingerprint*. Por ser determinístico, este tipo de campo permite comparações nos servidores. A mesma chave deve ser compartilhada entre os clientes e é usada tanto para cifrar quanto para decifrar os dados.
- Ordenável (OR): neste caso, o conteúdo do campo é cifrado por meio da função $encryptOR(key_{shared}, content)$, com um algoritmo de criptografia simétrica que preserva a relação de ordem nas cifras [Boldyreva et al. 2012, Boneh et al. 2014, Lewi and Wu 2016] e o resultado é utilizado como *fingerprint*. Este tipo de campo permite que servidores definam a relação entre duas tuplas comparando a ordem de seus campos, sem acessar o conteúdo das tuplas. A mesma chave deve ser compartilhada entre os clientes e é usada tanto para cifrar quanto para decifrar os dados.
- Operável (OP): para este campo é utilizado um par de chaves, sendo que o conteúdo do campo é cifrado por meio da função $encryptOP(key_{public}, content)$, utilizando um algoritmo homomórfico [Tourky et al. 2016] ou um parcialmente homomórfico [Naehrig et al. 2011] e o resultado é usado como *fingerprint*. Este tipo de campo permite operações nos servidores (ex.: somas e multiplicações) acessando apenas os conteúdos cifrados. Como operações podem ter sido executadas,

o *fingerprint* pode divergir da tupla a qual se refere, portanto, ao ler uma tupla e reconstruí-la através do esquema PVSS, o cliente deve atualizar o valor destes campos com os valores contidos no *fingerprint*. A chave privada deve ser compartilhada entre os clientes e é usada para decifrar os dados.
- Privado (PR): um símbolo especial é o *fingerprint*, i.e., o conteúdo deste campo é mantido cifrado sem qualquer possibilidade de realização de comparações ou acesso aos dados.

A Figura 1 apresenta a função usada para computar o *fingerprint* $t_h = \langle h_1, ..., h_m \rangle$ de uma tupla $t = \langle f_1, ..., f_m \rangle$, de acordo com a classificação apresentada. Vale destacar que o mesmo procedimento é usado nos *templates*, permitindo a verificação se um *template* combina com uma tupla [Bessani et al. 2008]. Esta classificação fornece a seguinte ordem para os campos, de acordo com o nível de segurança [Floriano et al. 2017a, Floriano et al. 2017b]: PU $<$ CO $<$ CD $<$ OR $<$ OP $<$ PR. Esta ordem está relacionada com a quantidade de informação revelada no *fingerprint* (ex.: a ordem entre os campos de duas tuplas).

$$h_i = \begin{cases} * & \text{if } f_i = * \\ f_i & \text{if } f_i \text{ is PU} \\ hash(f_i) & \text{if } f_i \text{ is CO} \\ encryptCD(key_{shared}, f_i) & \text{if } f_i \text{ is CD} \\ encryptOR(key_{shared}, f_i) & \text{if } f_i \text{ is OR} \\ encryptOP(key_{public}, f_i) & \text{if } f_i \text{ is OP} \\ \text{PR} & \text{if } f_i \text{ is PR} \end{cases}$$

Figura 1. Cálculo do *fingerprint*.

Finalmente, vale destacar que é necessário o compartilhamento e gerenciamento de chaves para que campos CD, OR e OP sejam usados. Estas chaves devem ser compartilhadas entre os clientes que desejam se comunicar. No caso de campos OP, os servidores também precisam da chave pública para executar operações. A Seção 4.1 apresenta um protocolo para compartilhamento de chaves através do próprio espaço de tuplas.

Controle de Acesso O controle de acesso é um mecanismo fundamental para manutenção da integridade e confidencialidade das informações (tuplas) armazenadas no DEPSPACE, pois previne que clientes não autorizados obtenham acesso as tuplas, além de impedir que clientes faltosos saturem o espaço de tuplas enviando uma grande quantidade de tuplas. O DEPSPACE implementa controle de acesso de duas formas:

- **Baseado em credenciais:** para cada tupla inserida no DEPSPACE pode-se definir quais são as credenciais necessárias para acessá-la, tanto para leitura quanto para remoção (acesso em nível de tuplas). Estas credenciais são definidas pelo processo que insere a tupla. Também é possível definir, quando o espaço de tuplas é criado, quais são as credenciais necessárias para inserir uma tupla no espaço (acesso em nível de espaço). A implementação desta funcionalidade é realizada através da associação de listas de controle de acesso a cada espaço e tupla.
- **Políticas de granularidade fina:** o DEPSPACE suporta a definição de políticas de acesso de granularidade fina [Bessani et al. 2006], que devem ser especificadas

no momento da criação do espaço de tuplas. Estas políticas controlam o acesso ao espaço e são definidas considerando três parâmetros: o identificador do cliente, a operação que será executada (juntamente com seus argumentos) e o estado atual do espaço de tuplas.

3.1. Implementação

A classificação anteriormente apresentada para geração do *fingerprint* é implementada no DEPSPACE através da utilização de esquemas robustos de criptografia, de acordo com cada campo.

- Público (PU): não utiliza criptografia uma vez que o seu conteúdo original é usado como *fingerprint*.
- Comparável (CO): estes campos utilizam o algoritmo SHA-1, que gera um *hash* de 20 *bytes*. Este campo é legado da implementação original do DEPSPACE e seu uso não é recomendado.
- Comparável Determinístico (CD): estes campos utilizam o algoritmo HMAC-SHA256 (*Hash-based Message Authentication Code with SHA-256*) e uma chave secreta de 256 *bits* para gerar uma saída pseudo-aleatória de 32 *bytes*.
- Operável (OP): estes campos utilizam a biblioteca *javallier* [Analytics 2017], que é uma implementação em Java do algoritmo de Paillier [Paillier 1999]. Esta biblioteca de criptografia parcialmente homomórfica permite a adição entre dois valores cifrados e através desta operação pode-se derivar a subtração. Adicionalmente, um valor cifrado pode ser multiplicado por um valor em claro usando repetidas operações de adição. Para este algoritmo assimétrico utilizamos um par de chaves (pública e privada) de 3072 *bits*, de modo a obter um nível de segurança de 128 *bits* [Alves and Aranha 2016].
- Ordenável (OR): para este tipo de campo foi utilizado o algoritmo de *Order Revealing Encryption* [Lewi and Wu 2016]. Este algoritmo baseado em AES, ao invés de preservar a ordem, gera textos cifrados não determinísticos que não podem ser comparados diretamente, mas sim submetidos a uma função de comparação que retorna a relação entre eles. Para este algoritmo foram utilizadas chaves simétricas de 128 *bits*. Este algoritmo, por fornecer resposta também de igualdade, dispensaria o uso dos tipos de campos CO e CD, visto que pode ser aplicado também a texto e não apenas a números, tornando o sistema ainda mais seguro por seu caráter não determinístico. Porém, aqueles campos ainda podem ser utilizados por questões de desempenho.
- Privado (PR): não utiliza criptografia uma vez que nenhuma informação é incluída no *fingerprint* (somente o símbolo PR).

4. Segurança e Privacidade em Coordenação Distribuída e Extensível

Esta seção discute como as soluções implementadas no DEPSPACE, principalmente os campos OP e OR que empregam esquemas robustos de criptografia e permitem a realização de computações sobre dados cifrados nos servidores, podem ser usadas na implementação de coordenação distribuída e extensível [Distler et al. 2015]. A ideia principal deste tipo de coordenação é diminuir a quantidade de comunicação necessária entre clientes e servidores, mas para isso é necessário que os servidores realizem processamentos sobre os dados armazenados.

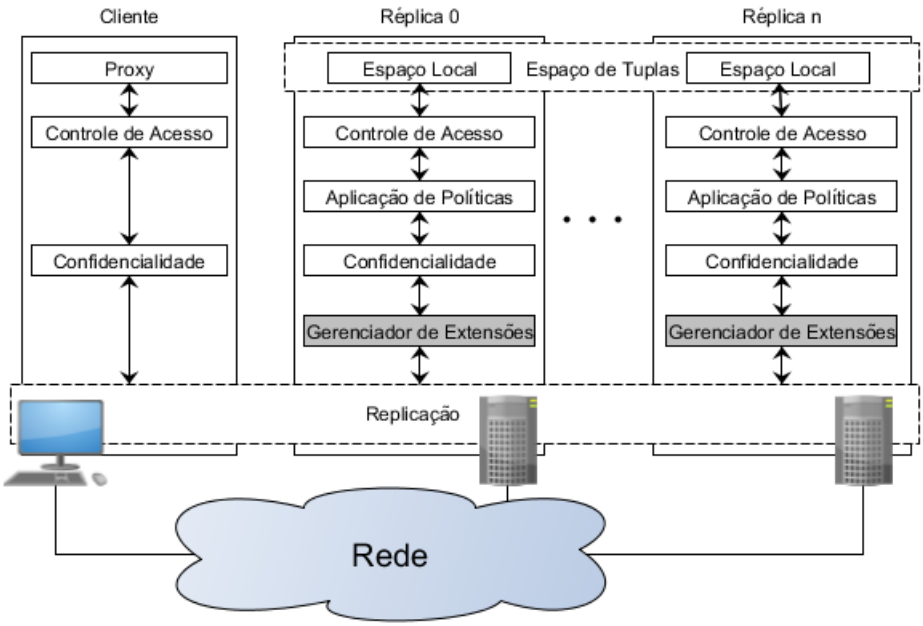


Figura 2. Camadas do DEPSPACE.

A Figura 2 apresenta as camadas do DEPSPACE, com uma camada adicional para gerenciamento de extensões [Distler et al. 2015]. Conforme já comentado, através da instalação de extensões nesta camada é possível que servidores realizem computações sobre campos OP e OR. Note que extensões seguras não podem ser implementadas sem a utilização destes campos, visto que dados em claro precisariam ser acessados pelos servidores.

Antes de uma extensão ser instalada, o gerenciador de extensões realiza uma série de verificações para determinar se este passo é seguro (ex.: uma extensão não pode conter laços infinitos de repetição ou recursividade). Depois que uma extensão é instalada, a mesma passa a interceptar as operações antes de serem processadas pelas camadas superiores e podem modificar o comportamento do sistema. No entanto, as mesmas devem apresentar um comportamento determinístico para manter a consistência no estado das réplicas e também são executadas de forma controlada, evitando por exemplo um consumo significativo de recursos (CPU), o acesso ao sistema de arquivos e a abertura de conexões [Distler et al. 2015].

As seções seguintes apresentam protocolos para implementação de dois importantes mecanismos de coordenação distribuída [Distler et al. 2015], que são os contadores compartilhados e as filas distribuídas. Por exemplo, estes mecanismos podem ser usados em *data centers* para coordenação e sincronização de processos. Para cada um dos mecanismos, são apresentadas e comparadas soluções que utilizam tanto o modelo tradicional de coordenação quanto o modelo extensível. Antes de apresentar estes protocolos, a seção seguinte apresenta um algoritmo para compartilhamento de chaves através do próprio espaço de tuplas visto que este gerenciamento é necessário para a utilização de campos CD, OR e OP.

4.1. Gerenciamento de Chaves

Gerenciamento de chaves não é uma tarefa trivial em sistemas distribuídos, e envolve desde o compartilhamento das chaves até a atualização das mesmas. O Algoritmo 1 apresenta um protocolo para compartilhamento de um par de chaves entre processos em um sistema distribuído, através de um espaço de tuplas. Algoritmo semelhante pode ser usado para compartilhar apenas uma chave secreta. Nas soluções propostas neste trabalho, este algoritmo deve ser usado antes que os campos CD, OR e OP possam ser utilizados (note que este algoritmo utiliza apenas campos PU e PR).

Algoritmo 1 Algoritmo para compartilhamento de chaves.

```
1:  void compartilharChaves() {
2:     my_key_private = gerar chave privada;
3:     my_key_public = gerar chave pública;
4:     t = ⟨"keys", my_key_public, my_key_private⟩ protegidos como ⟨PU, PU, PR⟩
5:     t̄ = ⟨"keys", *, *⟩ protegidos como ⟨PU, PU, PR⟩
6:     ⟨"keys", key'_pub, key'_priv⟩ = cas(t̄, t);
7:     if ⟨"keys", key'_pub, key'_priv⟩ != null then
8:        my_key_public = key'_pub
9:        my_key_private = key'_priv
10:    end if
11: }
```

A ideia por traz do algoritmo é que cada processo gere um par de chaves e tente disponibilizá-las no espaço para os outros processos. Para isso, utiliza a função cas com um *template* que combine com um par de chaves já disponível no espaço. Caso o retorno seja nulo, então estas foram as primeiras chaves a serem inseridas no espaço e o processo pode continuar a usá-las. Caso contrário, as chaves lidas do espaço passam a ser utilizadas. Como a chave privada fica em um campo PR, apenas os clientes que obtêm esta tupla do espaço conseguem acessá-la. Também deve-se ativar nestas tuplas os mecanismos de controle de acesso de forma que apenas os clientes autorizados, que fazem parte do grupo de coordenação, possam acessá-las. Por simplicidade, estas configurações de controle de acesso não são apresentadas nos algoritmos presentes neste trabalho. Finalmente, para atualizar esta chave ou o grupo de processos autorizados a acessá-la, basta inserir uma nova tupla no espaço com as configurações e dados apropriados. Para remover um processo do grupo, uma nova chave deve ser gerada (neste caso os *fingerprints* obtidos com a chave antiga devem ser atualizados). Por outro lado, a mesma chave pode ser usada para adicionar um processo no grupo.

4.2. Contador Compartilhado

Para implementar um contador compartilhado, os processos precisam ler e atualizar um contador armazenado em um tupla. O Algoritmo 2 apresenta a solução usando o modelo tradicional de coordenação. Neste algoritmo, para incrementar o contador, um processo precisa primeiro ler o valor armazenado no espaço (linha 4) para então incrementá-lo e atualizá-lo (linha 6). Neste modelo, caso haja concorrência no acesso ao contador, um processo pode não conseguir atualizar seu valor devido ao fato de outro processo já ter

Algoritmo 2 Contador compartilhado tradicional.

```
1:  int incrementar() {
2:     while true do
3:        t̄= ⟨cont, ∗⟩ protegidos como ⟨PU, CD⟩;          // define o template.
4:        ⟨cont, c⟩ = rdp (t̄);                            // lê o valor do contador
5:        cnt_novo = ⟨cont, c + 1⟩ protegidos como ⟨PU, CD⟩;   // define o novo valor
6:        ⟨cont, c⟩ = replace(⟨cont, c⟩,cnt_novo);          // tenta atualizar o contador
7:        if ⟨cont, c⟩ != null then   //termina em caso de sucesso, senão tenta novamente
8:           return c + 1;
9:        end if
10:    end while
11: }
```

realizado esta atualização. Neste caso, a função $replace$ (linha 6) retorna nulo e todo o procedimento precisa ser reiniciado.

Através da utilização de coordenação extensível, o Algoritmo 3 diminui os passos de comunicação necessários para realizar a operação de incremento (apenas um rdp ao invés de um rdp e um $replace$ do Algoritmo 2) e elimina a necessidade de reprocessamentos devido a acessos concorrentes. Neste caso, uma extensão (linhas 6-13) é instalada nos servidores e quando uma operação rdp é entregue através dos protocolos da replicação máquina de estados, o servidor incrementa o valor do contador (para isso este campo é configurado como OP) e retorna o valor já atualizado. Nestes algoritmos, o prefixo "local" (linha 8) é utilizado para indicar um operação local nos servidores, que não envolve o envio de mensagens (execuções de RPCs). Note que as garantias de terminação também são diferentes, sem extensões é garantido que um cliente termina apenas se outros não estiverem acessando o sistema (*obstruction freedom*) enquanto que com extensões os clientes sempre terminam (*wait freedom*).

Algoritmo 3 Contador compartilhado usando coordenação extensível.

Cliente:

```
1:  int incrementar() {
2:     t̄= ⟨cont, 1⟩ protegidos como ⟨PU, OP⟩;          // define o template.
3:     ⟨cont, c⟩ = rdp (t̄);                            // lê o valor já atualizado do contador
4:     return c;
5:  }
```

Extensão nos servidores:

```
6:  Tuple rdp(⟨cont, cif_recebido⟩) {
7:     default = ⟨cont, ∗⟩ protegidos como ⟨PU, OP⟩;
8:     ⟨cont, cif_atual⟩ = local.rdp(default);  //busca localmente o valor atual do contador
9:     cif_novo = soma(cif_atual, cif_recebido)          //operação possível por ser campo OP
10:    cnt_novo = ⟨cont, cif_novo⟩ protegidos como ⟨PU, OP⟩;
11:    local.replace(⟨cont, cif_atual⟩, cnt_novo);
12:    return cnt_novo;
13: }
```

4.3. Fila Distribuída

Para implementar uma fila distribuída através de um espaço de tuplas, os processos precisam inserir tuplas no espaço representando os elementos da fila. Cada elemento contém um identificador que será usado para organizar os elementos em ordem na fila. Neste trabalho, utilizamos o tempo da criação dos elementos para esta finalidade. Posteriormente, para remover o primeiro elemento da fila, um processo precisa remover a tupla com o menor tempo de criação.

O Algoritmo 4 apresenta a solução que utiliza o modelo tradicional. Para inserir um elemento na fila basta inserir a tupla representando este elemento no espaço. No entanto, na remoção primeiramente todas as tuplas com elementos precisam ser lidas pelo processo (linha 8) que as organiza segundo o tempo de criação para então tentar remover o primeiro elemento da fila (linha 11). Esta função pode retornar nulo devido ao acesso concorrente de outro processo que já removeu o elemento em questão. Desta forma, aumentar a concorrência também aumenta as necessidades de reprocessamentos.

Algoritmo 4 Fila distribuída tradicional.

```
 1: void add(ElementoId eid, byte[] dados) {
 2:    t = ⟨fila, eid, dados⟩ protegidos como ⟨PU, CD, PR⟩
 3:    out(t);
 4: }

 5: byte[] remove() {
 6:    while true do
 7:       t̄= ⟨fila, ∗, ∗⟩ protegidos como ⟨PU, CD, PR⟩;        // define o template.
 8:       Tuple[] tuples = rdAll (t̄);                          // lê os elementos
 9:       Ordenar as tuplas em tuples pelos seus ids (segundo campo)
10:       for all t in tuples do
11:          ⟨fila, id, dados⟩ = inp(t);        //tenta remover o primeiro elemento da fila
12:          if ⟨fila, id, dados⟩ != null then     //termina caso conseguir remover a tupla
13:             return dados;
14:          end if
15:       end for
16:    end while
17: }
```

O Algoritmo 5 apresenta a solução usando coordenação extensível. Assim como no contador, esta solução diminui os passos de comunicação e elimina a necessidade de reprocessamentos. A inserção de um elemento funciona como no modelo anterior, já na remoção uma extensão é instalada nos servidores para remover e retornar o primeiro elemento da lista. Para isso, os campos com os identificadores dos elementos devem ser configurados como OR.

5. Experimentos

Visando analisar o desempenho dos mecanismos propostos para coordenação distribuída, bem como avaliar as melhorias no desempenho através do uso de soluções para coordenação extensível, os algoritmos anteriormente propostos foram implementados no DEPS-

Algoritmo 5 Fila distribuída usando coordenação extensível.

Cliente:

```
1:  void add(ElementoId eid, byte[] dados) {
2:      t = ⟨fila, eid, dados⟩ protegidos como ⟨PU, OR, PR⟩
3:      out(t);
4:  }

5:  byte[] remove() {
6:      t̄= ⟨fila, ∗, ∗⟩ protegidos como ⟨PU, OR, PR⟩
7:      ⟨fila, id, dados⟩ = inp(t̄);
8:      return dados;
9:  }
```

Extensão nos servidores:

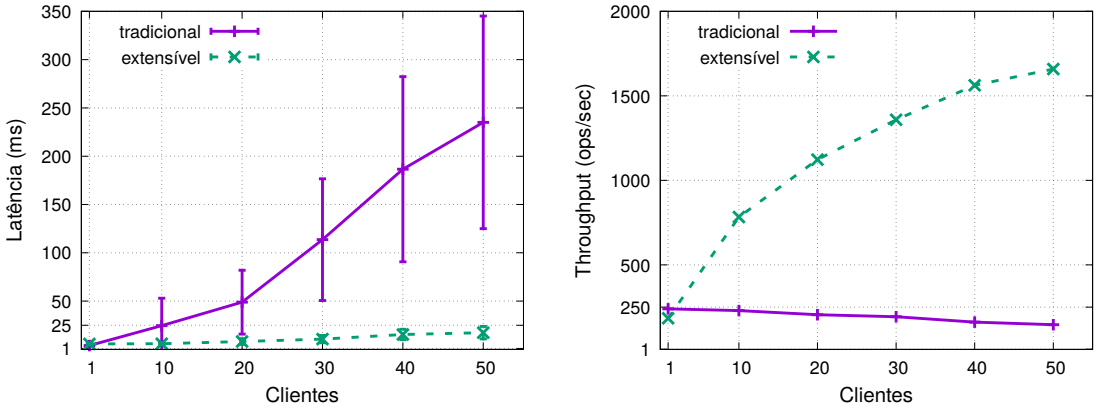
```
10: Tuple inp(⟨fila, ∗, ∗⟩) {
11:     default = ⟨fila, ∗, ∗⟩ protegidos como ⟨PU, OR, PR⟩;
12:     Tuple[] tuples = local.rdAll(default);          // lê localmente os elementos
13:     head = tupla em tuples com menor id  //comparação possível por ser campo OR
14:     local.inp(head);
15:     return head;
16: }
```

PACE e alguns experimentos foram realizados no Emulab [White et al. 2002]. O principal objetivo destes experimentos não é determinar os valores máximos de desempenho, mas sim analisar as diferenças entre as abordagens e determinar os ganhos advindos com as soluções que permitem computações nos servidores.

Configuração dos Experimentos. O ambiente para os experimentos foi constituído por 6 máquinas *d430* (2.4 GHz E5-2630v3, com 8 núcleos e 2 *threads* por núcleo, 64GB de RAM e interface de rede gigabit) conectadas a um *switch* de 1Gb. O DEPSPACE foi configurado com 4 servidores para tolerar até uma falha maliciosa. Cada servidor foi executado em uma máquina separada, enquanto que os clientes foram distribuídos uniformemente nas outras 2 máquinas. O ambiente de *software* utilizado foi o sistema operacional Ubuntu 14 64-*bit* e máquina virtual Java de 64 *bits* versão 1.8.0_131.

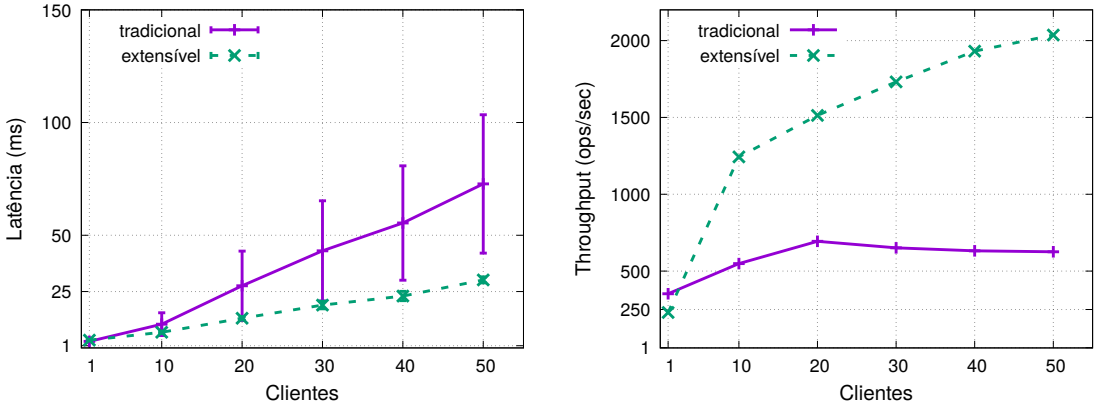
Resultados e Análises. Para verificar o desempenho das diferentes abordagens, variamos a quantidade de clientes e medimos a vazão (*throughput*) e a latência apresentadas pelo sistema. A vazão foi medida nos servidores a cada 1000 requisições processadas, enquanto que a latência foi medida nos clientes em uma execução com 1000 repetições. Os gráficos desta seção apresentam os valores médios obtidos para estas métricas. Nos experimentos com a fila distribuída, cada cliente primeiramente adicionava um dado (como elemento da fila, usamos um vetor com zero *bytes* para avaliar apenas os custos relacionado com as interações no protocolo) e depois removia o primeiro elemento da fila, garantindo que a mesma nunca estivesse vazia no momento da remoção.

A Figura 3 apresenta os valores para o desempenho dos protocolos para o contador compartilhado, enquanto que a Figura 4 apresenta os valores para a fila distribuída. De



(a) Latência (b) *Throughput*

Figura 3. Latência e vazão (*throughput*) do contador compartilhado.



(a) Latência (b) *Throughput*

Figura 4. Latência e vazão (*throughput*) da fila distribuída.

uma forma geral, as soluções que permitem processamento nos servidores (extensíveis) apresentam um desempenho significativamente superior ao das soluções tradicionais. De fato, conforme o número de clientes aumenta, a concorrência para acesso aos dados compartilhados aumenta e a quantidade de reprocessamentos (devido a falhas no incremento do contador ou na remoção do início da fila) também se torna maior. Este comportamento acaba impactando negativamente o desempenho do sistema em sua forma tradicional. Entretanto, na versão extensível o desempenho é significativamente melhor utilizando processamentos nos servidores mesmo que para isso precise realizar computações criptográficas mais custosas (campos OP ou OR).

Os experimentos também mostraram que o desvio padrão para a latência das soluções tradicionais é muito grande, o que é explicado pelo fato de algumas operações não precisarem de reprocessamentos enquanto que outras podem repetir várias vezes até conseguirem realizar as computações desejadas (incrementar o contador ou remover o início da lista). Finalmente, vale destacar que a diferença entre as duas abordagens é menor na fila do que no contador porque a operação de inserção na fila funciona da mesma forma para as duas abordagens.

6. Conclusões

Este trabalho apresentou uma proposta para implementação de mecanismos de coordenação distribuída com propriedades de segurança e privacidade. Através da utilização de esquemas robustos de criptografia, que permitem a realização de processamentos sobre dados cifrados nos servidores, foram propostos protocolos para coordenação distribuída e extensível, os quais apresentam um desempenho significativamente superior aos métodos tradicionais de coordenação, principalmente por demandarem um número menor de acessos ao espaço de tuplas (diminuindo a quantidade de RPCs) e eliminarem a necessidade de reprocessamentos visto que os servidores já executam todas as ações necessárias, de acordo com a operação desejada e o mecanismo de coordenação utilizado.

Referências

Alves, P. G. M. R. and Aranha, D. F. (2016). A framework for searching encrypted databases. In *XVI Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais (SBSEG 2016)*, pages 142–155. SBC.

Analytics, N. (2017). A java library for paillier partially homomorphic encryption. GitHub. `https://github.com/n1analytics/javallier`.

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

Bakken, D. E. and Schlichting, R. D. (1995). Supporting Fault-Tolerant Parallel Programing in Linda. *IEEE Transactions on Parallel and Distributed Systems*, 6(3):287–302.

Baldoni, R., Marchetti, C., and Verde, L. (2003). Corba request portable interceptors: Analysis and applications. *Concurrency and Computation: Practice and Experience*, 15(6):551–579.

Bershad, B. N., Savage, S., Pardyak, P., Sirer, E. G., Fiuczynski, M. E., Becker, D., Chambers, C., and Eggers, S. (1995). Extensibility safety and performance in the spin operating system. In *Proceedings of 15th Symposium on Operating Systems Principles*.

Bessani, A., Alchieri, E., Correia, M., and da Silva Fraga, J. (2008). DepSpace: A byzantine fault-tolerant coordination service. *European Conference on Computer Systems*.

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

Bessani, A. N., Correia, M., Fraga, J. S., and Lung, L. C. (2006). Sharing memory between Byzantine processes using policy-enforced tuple spaces. In *Proceedings of 26th IEEE International Conference on Distributed Computing Systems - ICDCS 2006*.

Boldyreva, A., Chenette, N., Lee, Y., and O'Neill, A. (2012). Order-preserving symmetric encryption. Cryptology ePrint Archive, Report 2012/624.

Boneh, D., Lewi, K., Raykova, M., Sahai, A., Zhandry, M., and Zimmerman, J. (2014). Semantically secure order-revealing encryption: Multi-input functional encryption without obfuscation. Cryptology ePrint Archive, Report 2014/834.

Boneh, D. and Shoup, V. (2015). A graduate course in applied cryptography. `https://crypto.stanford.edu/~dabo/cryptobook/draft_0_2.pdf`.

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

Distler, T., Bahn, C., Bessani, A., Fischer, F., and Junqueira, F. (2015). Extensible distributed coordination. In *Proc. of 10th European Conference on Computer Systems*.

Floriano, E., Alchieri, E., Aranha, D., and Solis, P. (2017a). Privacidade em dados armazenados em memória compartilhada através de espaços de tupla. In *Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*.

Floriano, E., Alchieri, E., Aranha, D., and Solis, P. (2017b). Providing privacy on the tuple space model. *Journal of Internet Services and Applications*, 8(19):1–16.

Gelernter, D. (1985). Generative Communication in Linda. *ACM Transactions on Programing Languages and Systems*, 7(1):80–112.

Hunt, P., Konar, M., Junqueira, F. P., and Reed, B. (2010). Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, pages 11–11.

Lewi, K. and Wu, D. J. (2016). Order-revealing encryption: New constructions, applications, and lower bounds. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1167–1178.

Naehrig, M., Lauter, K., and Vaikuntanathan, V. (2011). Can homomorphic encryption be practical? In *Proceedings of 3rd Workshop on Cloud Computing Security Workshop*.

Naveed, M., Kamara, S., and Wright, C. V. (2015). Inference attacks on property-preserving encrypted databases. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 644–655.

Paillier, P. (1999). Public-key cryptosystems based on composite degree residuosity classes. In *Proceedings of the 17th International Conference on Theory and Application of Cryptographic Techniques*, EUROCRYPT'99, pages 223–238.

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

Schoenmakers, B. (1999). A simple publicly verifiable secret sharing scheme and its application to electronic voting. In *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology - CRYPTO'99*, pages 148–164.

Segall, E. J. (1995). Resilient distributed objects: Basic results and applications to shared spaces. In *Proceedings of the 7th Symposium on Parallel and Distributed Processing*.

Tourky, D., ElKawkagy, M., and Keshk, A. (2016). Homomorphic encryption the "holy grail" of cryptography. In *2nd IEEE Conference on Computer and Communications*.

Veríssimo, P. (2016). Dialogue on cyber policies between brazil and the eu: prospecting threats and opportunities of the cyberspace. Dialogue on Cyber Policies.

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