Um Espaço de Tuplas Tolerante a Intrusões sobre P2P

Davi da Silva Böger[1], **Eduardo Alchieri**[2], **Joni da Silva Fraga**[1]

[1]DAS - Universidade Federal de Santa Catarina (UFSC)
[2]CIC, Universidade de Brasília (UnB)

Resumo. *O uso de espaços de tuplas tem se mostrado uma solução atrativa para coordenação entre processos em sistemas distribuídos abertos e dinâmicos, como redes P2P, principalmente pelas suas características de desacoplamento espacial e temporal. Nestes ambientes, caracterizados como sistemas heterogêneos e abertos, aumenta significativamente a possibilidade de processos maliciosos estarem presentes em determinada computação. Neste sentido, este trabalho apresenta nossos esforços na concretização de um Espaço de Tuplas (ET) sobre um overlay P2P, que tolera a presença de processos maliciosos. O ET é construído sobre uma infraestrutura para construção de memórias compartilhadas dinâmicas e tolerantes a intrusões, descrita em outro trabalho.*

1. Introdução

As redes par a par (**peer-to-peer**, P2P) são uma arquitetura interessante para sistemas distribuídos, pois permitem o uso eficiente dos recursos ociosos disponíveis na Internet e têm a capacidade de aumentar o número de nós sem detrimento do desempenho. Algumas redes P2P oferecem primitivas de comunicação com latência e número de mensagens de ordem logarítmica em relação ao número de nós [Rowstron and Druschel 2001]. Essa escalabilidade permite que recursos disponíveis em uma grande quantidade de nós possa ser utilizado de maneira vantajosa.

Apesar das vantagens, as redes P2P apresentam desafios para o provimento de confiabilidade. Essas redes normalmente são formadas dinamicamente por nós totalmente autônomos que podem entrar e sair do sistema a qualquer momento. Tal dinamismo dificulta a manutenção da consistência das informações distribuídas no sistema, bem como a construção de aplicações mais complexas que poderiam se beneficiar da escalabilidade [Baldoni et al. 2005]. A grande maioria dos sistemas P2P são aplicações de disseminação de informações pouco mutáveis ou autoverificáveis. Além disso, essas redes não possuem gerência global, são redes de pares com grande abertura. Logo, as redes P2P podem conter nós maliciosos que colocam em risco o funcionamento das aplicações [Wallach 2003].

Um Espaço de Tuplas [Gelernter 1985] (ET) é um mecanismo de coordenação de processos que apresenta desacoplamento espacial, isto é, processos não precisam conhecer os endereços um do outro para se comunicar; e desacoplamento temporal, isto é, processos não precisam estar ativos simultaneamente para efetivar a comunicação. Essas propriedades são garantidas pelo uso de uma memória associativa na qual as mensagens persistem e podem ser acessadas por buscas baseadas no conteúdo. Essas mensagens persistentes e endereçáveis por conteúdo são denominadas **tuplas** e a memória que as armazena é denominada de **espaço de tuplas**. Os espaços de tuplas são especialmente interessantes em sistemas abertos e dinâmicos, como as redes P2P, nos quais processos não possuem conhecimento completo sobre os demais participantes do sistema. Nós podem se comunicar armazenando tuplas em um ET global que garante a persistência dessas informações, mesmo após a saída do nó que produziu a tupla.

Neste sentido, este trabalho descreve a construção de um espaço de tuplas tolerante a intrusões, que executa sobre uma rede P2P. O espaço de tuplas funciona sobre uma infraestrutura para suporte a aplicações de memória compartilhada tolerantes a intrusões definida em [Böger et al. 2012]. Essa infraestrutura divide um **overlay** P2P estruturado em um conjunto de segmentos e fornece operações para encontrar e acessar segmentos, que são dinâmicos e suportam a entrada e saída de nós, executando um protocolo de Replicação Máquina de Estados (RME) reconfigurável [Lamport et al. 2010].

Para que o ET proposto possa fazer uso dessa segmentação, é necessário cumprir com a exigência relacionada à indexação do estado da aplicação, que deve ser feita de forma a permitir a divisão e união de partes do estado de acordo com as divisões e uniões dos segmentos do sistema. De maneira geral, um espaço de tuplas é um multiconjunto de tuplas, isto é, uma coleção de tuplas que permite repetição de elementos idênticos. Indexamos o ET atribuindo uma chave a cada tupla e dividimos o mesmo em espaços menores contendo parte das tuplas com chaves numericamente próximas. Cada segmento S, conforme definido na camada de segmentação, armazena as tuplas com chaves dentro do intervalo $K(S)$. Para atribuição das chaves, utilizamos um esquema de indexação multidimensional baseado em **Curvas de Preenchimento de Espaço** (**Space Filling Curves** ou SFCs em inglês), em especial a **Curva de Hilbert** [Lawder and King 2000]. A Curva de Hilbert apresenta a propriedade de ter boa localidade, isto é, tuplas com conteúdo próximo tendem a possuir índices próximos, o que facilita a realização de buscas por conteúdo.

O ET proposto neste artigo, por ser baseado na infraestrutura descrita em [Böger et al. 2012], apresenta a propriedade de garantir a consistência das informações mesmo na presença de nós maliciosos. Isso assegura o funcionamento correto do ET, porém não provê características de segurança como privacidade de tuplas e controle de acesso. Existem maneiras de garantir essas propriedades de segurança adicionais [Bessani et al. 2008], porém isso está fora do escopo deste trabalho.

O restante do artigo está dividido da seguinte forma: a Seção 2 apresenta o conceito de espaços de tuplas; a Seção 3 apresenta o modelo de sistema; a Seção 4 descreve as curvas de preenchimento de espaço e a curva de Hilbert; a Seção 5 apresenta a construção do espaço de tuplas propriamente dito e a Seção 6 discute as soluções propostas; finalmente, a Seção 7 aborda alguns trabalhos relacionados e a Seção 8 conclui o artigo.

2. Espaços de Tuplas

Um espaço de tuplas (ET) pode ser visto como um objeto de memória compartilhada que permite a interação entre processos distribuídos [Gelernter 1985]. Neste espaço, estruturas de dados genéricas chamadas de **tuplas**, podem ser inseridas, lidas e removidas. Uma tupla t é uma sequência ordenada de campos $\langle f_1 ... f_n \rangle$, onde cada campo f_i que contém um valor é dito *definido*. Uma tupla onde todos os campos são definidos é chamada de **entrada**. Uma tupla $\bar{t}$ é chamada de **molde** se alguns de seus campos não possuem valores definidos. Um espaço de tuplas somente pode armazenar entradas, nunca moldes. Os moldes são usados para acessar as tuplas do espaço. Diz-se que uma entrada t e um molde $\bar{t}$ **combinam** ($t \leq_m \bar{t}$) se e somente se: (i) ambos têm o mesmo número de campos, e (ii) todos os valores dos campos definidos em t possuem o mesmo valor dos campos correspondentes em $\bar{t}$. Por exemplo, uma tupla $\langle \text{WTF}, \text{BSB}, 2013 \rangle$ combina com o molde $\langle \text{WTF}, *, 2013 \rangle$ (onde '$*$' denota um campo não definido do molde).

Um espaço de tuplas provê três operações básicas [Gelernter 1985]: $out(t)$ que

adiciona uma tupla t no espaço de tuplas; $in(\bar{t})$ que lê e remove, do espaço de tuplas, uma tupla t que combine com o molde $\bar{t}$; e $rd(\bar{t})$ que tem um comportamento similar ao da operação in, mas que somente faz a leitura da tupla combinando com $\bar{t}$, sem removê-la do espaço. As operações in e rd são bloqueantes, i.e., se nenhuma tupla que combine com o molde $\bar{t}$ está disponível no espaço de tuplas, o processo fica bloqueado até que uma se faça disponível. Uma extensão típica deste modelo é a provisão de variantes não bloqueantes das operações de leitura: inp e rdp [Gelernter 1985]. Estas operações funcionam exatamente como suas versões bloqueantes, a não ser pelo fato que retornam um valor de erro quando uma tupla que combine com o molde não esteja disponível no espaço de tuplas. Uma característica importante do espaço de tuplas é a natureza associativa do acesso: as tuplas não são acessadas por um endereço ou identificador, mas sim pelo seu conteúdo.

3. Modelo de Sistema

Consideramos um sistema distribuído formado por um conjunto infinito Π de processos (ou nós), interconectados por enlaces de comunicação (**links**) formando desta maneira uma rede. Cada nó possui um endereço único de rede e pode enviar mensagens para qualquer outro nó, desde que conheça seu endereço. Um nó correto sempre age de acordo com a especificação dos seus protocolos. Um nó malicioso (ou bizantino [Lamport et al. 1982]) não se comporta segundo as especificações, agindo de maneira arbitrária. Assume-se que em qualquer momento da execução, no máximo f nós faltosos estão presentes no sistema. Este parâmetro é global e conhecido por todos os nós.

A infraestrutura subjacente ao espaço de tuplas, definida em [Böger et al. 2012], está dividida em quatro camadas, conforme ilustrado na Figura 1. A camada inferior é a rede, que oferece canais ponto a ponto confiáveis entre quaisquer pares de nós. O atraso na entrega das mensagens e as diferenças de velocidades entre os nós do sistema respeitam um modelo de sincronia parcial [Dwork et al. 1988], no qual é garantido a terminação de protocolos de Replicação Máquina de Estados que são usados nas camadas superiores. No entanto, não há garantia de sincronismo por toda a execução. Imediatamente acima da rede, encontram-se duas camadas independentes que são usadas para construir a camada de segmentação: a camada de **overlay** implementa uma rede P2P estruturada, no estilo anel (p. ex. [Rowstron and Druschel 2001]), sobre a camada de rede, com busca de nós distribuídos eficiente e tolerante a intrusões (em [Böger et al. 2012], o **overlay** definido em [Castro et al. 2002] é utilizado); e a camada de suporte à replicação fornece uma abstração de Replicação Máquina de Estados (RME) reconfigurável (em [Böger et al. 2012], a estratégia definida em [Lamport et al. 2010] é assumida) usada para garantir a disponibilidade e consistência das informações contidas no sistema. Em [Böger et al. 2012], tanto as premissas de sistema quanto as funcionalidades das camadas inferiores são apresentados em maior detalhe.

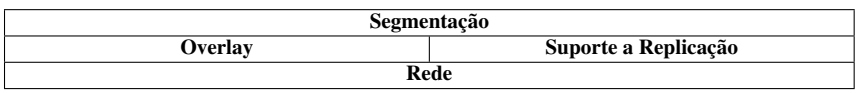

Figura 1. Camadas do sistema.

A camada de segmentação divide o anel lógico do **overlay** em segmentos compostos de nós contíguos, sendo cada segmento responsável por um intervalo de chaves do **overlay**. Para fins de disponibilidade, todos os nós do mesmo segmento armazenam o

mesmo conjunto de dados replicados. A consistência desses dados é mantida usando Replicação Máquina de Estados reconfigurável, provido pela camada de suporte à replicação.

Os segmentos são dinâmicos, ou seja, suas composições podem mudar com o tempo a partir da entrada e saída de nós. Cada segmento é descrito por um certificado de segmento (S), que contém a lista dos nós membros ($S.members$), o intervalo de chaves do **overlay** que cabe ao segmento ($K(S) = [S.start, S.end)$) e um contador de reconfigurações ($S.confId$). A cada reconfiguração, um novo conjunto de nós (denominado composição ou visão) passa a participar no segmento e a executar os algoritmos associados de suporte à RME, bem como um novo certificado de segmento é gerado e assinado pelos membros do segmento antigo, garantindo a autenticidade do mesmo. O número de nós de um segmento pode aumentar ou diminuir, porém, para manter o desempenho da RME, esse número é mantido dentro de um certo intervalo $[n_{MIN}, n_{MAX}]$. Isso é garantido pela divisão de segmentos grandes e pela união de segmentos pequenos adjacentes.

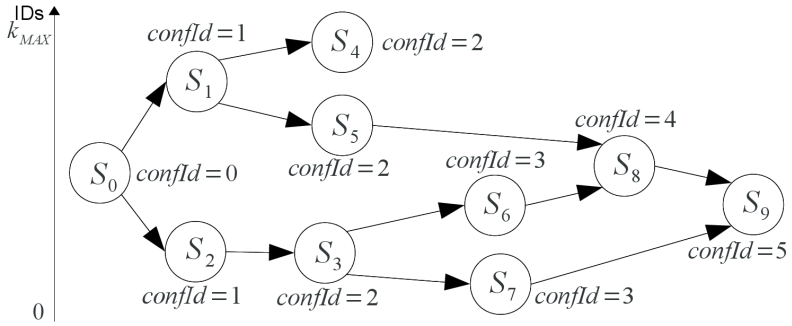

Figura 2. Dinâmica da segmentação em uma possível execução

A Figura 2 ilustra as reconfigurações ocorridas em uma possível execução. Com base nessa execução, três situações de reconfiguração distintas são descritas a seguir:

(1) O segmento inicial S_0 inicia a execução sendo responsável por todas as chaves do **overlay**, ou seja, $K(S_0) = [0, k_{MAX}]$, onde k_{MAX} é a maior chave possível. S_0 possui $confId = 0$, indicando que é o segmento inicial. Quando o número de nós de S_0 aumenta para além de n_{MAX}, ocorre a divisão, na qual são gerados os segmentos S_1 e S_2, ambos com $confId = 1$. Os membros de S_0, juntamente com os membros que recém entraram no sistema, são divididos entre os novos segmentos, de forma que S_1 e S_2 possuam ambos um número de nós maior ou igual a n_{MIN};

(2) A partir do segmento S_2, se um certo número de nós entrar e sair do segmento de forma que o número de nós se mantém estável, a reconfiguração não precisa realizar união ou divisão. Dessa forma, o conjunto de chaves cobertas pelo segmento não se altera pela reconfiguração, ou seja, $K(S_3) = K(S_2)$. O parâmetro $confId$ é incrementado, de forma que $S_3.confId = S_2.confId + 1 = 2$;

(3) Partindo do segmento S_6, pode ocorrer de uma parcela grande dos nós do segmento sair do sistema. Isso leva S_6 a iniciar uma união a fim de evitar terminar a reconfiguração com menos de n_{MIN} nós. Para tal é utilizado o segmento sucessor de S_6, isto é, o segmento S_5. O conjunto de membros resultantes da união é formado pelos membros dos dois segmentos, levando em conta pedidos de entrada e saída de ambos. O conjunto de chaves do segmento resultante S_8 é dado pela união dos conjuntos de chaves de S_5 e S_6, ou seja, $K(S_8) = K(S_5) \cup K(S_6)$. O valor de $confId$ é o incremento do maior valor nos segmentos S_5 e S_6, de forma que $S_8.confId = \max\{S_5.confId, S_6.confId\} + 1 = 4$.

A camada de segmentação apresenta as seguintes operações:

• $SegJoin(C_p)$ realiza a entrada do nó p na camada de segmentação por meio da apresentação do certificado de nó C_p. Esse certificado é o mesmo usado na camada de **overlay**;

• $SegLeave(C_p)$ trata da saída do nó p. C_p é o certificado apresentado na entrada;

• $SegFind(k, k')$ busca todos os certificados de segmento responsáveis por chaves no intervalo $[k, k']$. Os certificados de segmentos encontrados são repassados à camada superior por meio da chamada $SegFindOk$. A operação garante que todo o intervalo de busca é coberto pelos segmentos encontrados;

• $SegRequest(S_q, req)$ envia a requisição req ao segmento S_q usando a camada de suporte à replicação e retorna a resposta recebida da ME. Um temporizador é usado para interromper a requisição caso S_q seja obsoleto. Nesse caso, a operação retorna um valor distinto $\perp$. Caso contrário, a operação $SegDeliver(C_p, req)$ entrega req conforme a ME do segmento de destino. Atrasos na rede podem provocar o estouro do temporizador, mesmo se S_q for atual. Isso deve ser tratado na camada superior;

• $SegResponse(C_p, resp)$ envia a resposta ao cliente através da camada de rede;

• $SegNotify$ é bastante simples, consistindo no envio direto de uma mensagem a um nó, utilizando para isso a camada de rede. No nó cliente, qualquer mensagem recebida de pelo menos $f + 1$ nós de um mesmo segmento é repassada com $SegNotifyDeliver$;

• $SegReconfigure()$ operação interna da camada de segmentação, executada em intervalos de tempo determinados, que efetiva as entradas/saídas de nós conforme as chamadas de $SegJoin$/$SegLeave$. Nessa operação uniões e divisões de segmentos são efetivadas;

• $SegGetAppState$ e $SegSetAppState$ servem, respectivamente, para obter e alterar o estado da aplicação que executa acima da camada de segmentação.

4. Curvas de Preenchimento de Espaço

Uma curva de preenchimento de espaço (**Space Filling Curve**, SFC) é um mapeamento bidirecional entre pontos de um hipercubo de D dimensões e pontos de uma linha, ou seja, é uma curva que passa por todos os pontos de um hipercubo D-dimensional exatamente uma vez [Lawder and King 2000]. A Figura 3 (b) mostra um exemplo simples de uma SFC sobre um quadrado duas dimensões e coordenadas entre $(0, 0)$ e $(3, 3)$. Cada ponto do espaço é representado por um quadrado e a curva passa exatamente uma vez no centro de cada um desses quadrados.

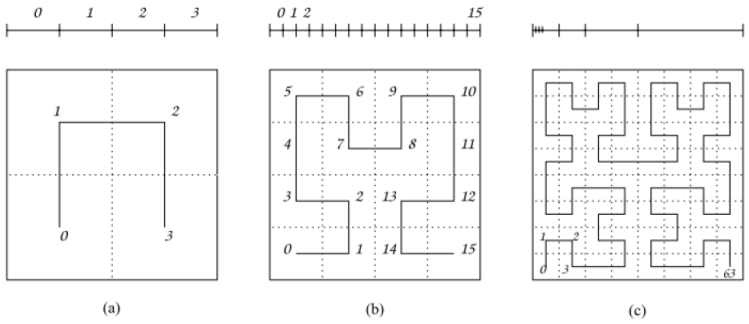

Figura 3. Desenho da curva de Hilbert bidimensional nas três primeiras ordens

Uma SFC pode ser usada como mecanismo de indexação de tuplas em um ET. Podemos tratar as tuplas como elementos de um espaço multidimensional, cada campo da tupla sendo uma coordenada, e podemos usar uma SFC para calcular um índice (ou

chave) associado à tupla. Usando o exemplo da Figura 3 (b), podemos indexar tuplas de dois campos com valores numéricos entre 0 e 3: a tupla $\langle 0, 0\rangle$ tem índice 0, a tupla $\langle 1, 0\rangle$ tem índice 1, a tupla $\langle 1, 1\rangle$ tem índice 2, e assim sucessivamente. Esse exemplo é simplificado e não permite a indexação de tuplas em situações mais complexas. A Seção 5 mostra uma generalização desse princípio que permite indexação de qualquer tupla.

Dado um mesmo espaço multidimensional, existem diversas formas de se indexar os pontos desse espaço, ou seja, existe mais de uma SFC que pode ser aplicada, e cada indexação apresenta propriedades distintas. A curva de Hilbert é um tipo de SFC que apresenta propriedades de localidade, no sentido de que pontos adjacentes na curva unidimensional são também adjacentes no espaço multidimensional [Lawder and King 2000]. Essa localidade facilita as buscas por conteúdo necessárias em um ET, uma vez que tuplas que casem com o mesmo molde tem chance maior de terem índices próximos. Usando ainda o exemplo da Figura 3 (b), podemos ver que uma busca usando o molde $\langle 1, *\rangle$ precisa localizar tuplas com índices nos intervalos $[1, 2]$ e $[6, 7]$.

De maneira geral, dado um espaço multidimensional de D dimensões, uma curva de Hilbert de ordem k divide o espaço em 2^{Dk} subespaços iguais e trata o centro desses subespaços como os pontos pelos quais a curva deve passar. Dessa forma, uma curva de ordem k divide cada eixo do espaço em k níveis e possui 2^{Dk} pontos. A Figura 3 ilustra as três primeiras ordens de uma curva de Hilbert em duas dimensões. A curva de primeira ordem (a) divide o quadrado em quatro partes e percorre os quadrantes em sentido horário. A curva de segunda ordem (b) é construída a partir da curva de primeira ordem pela substituição de cada quadrante por uma curva de primeira ordem rotacionada. A curva de terceira ordem (c) é obtida de maneira similar. Em geral, a curva de ordem $k + 1$ é construída a partir da curva de ordem k dividindo cada ponto desta curva em um subespaço de 2^D pontos e traçando uma curva de ordem 1 rotacionada de forma a se conectar com os subespaços vizinhos. Mais detalhes sobre o procedimento de cálculo podem ser encontrados em [Lawder and King 2000].

A curva completa é obtida aplicando a definição recursiva infinitamente, porém para a construção de índices são usadas as curvas de ordem finita. A ordem da curva está relacionada à quantidade de informação (número de bits) necessária para representar as coordenadas dos pontos do espaço e dos pontos da curva unidimensional. Em geral, dado um espaço D-dimensional e uma curva de k-ésima ordem, cada coordenada dos pontos do espaço precisam de k bits e um ponto qualquer (tanto no espaço multidimensional quanto na curva unidimensional) precisa de $D \times k$ bits. Assumimos que, dado um ponto x no espaço multidimensional, $Hilbert(x)$ representa o ponto na curva de Hilbert correspondente a x.

5. Espaço de Tuplas Tolerante a Intrusões

A camada de espaço de tuplas, construída sobre a camada de segmentação, implementa as operações out, rd, rdp, in e inp. A indexação do espaço de tuplas, necessária para a camada de segmentação, é realizada pela atribuição de chaves a tuplas individuais e de conjuntos de chaves a moldes. De maneira geral, se uma tupla t possui chave k, então qualquer molde $\bar{t}$, tal que $t \leq_m \bar{t}$, deve possuir um conjunto de chaves $\bar{K}$ tal que $k \in \bar{K}$. Dessa maneira, se a tupla t é armazenada no segmento responsável por k, então uma busca cobrindo $\bar{K}$ termina por encontrar t. Esta estratégia não necessita que chaves associadas às tuplas sejam únicas, i.e., tuplas diferentes podem possuir a mesma chave. Basta que a chave associada à tupla aponte sempre para o segmento que a armazena a tupla.

A Curva de Hilbert é utilizada para atribuir chaves a tuplas e conjuntos de chaves a moldes, de forma a possibilitar que as tuplas inseridas no ET sejam posteriormente encontradas. Seja D um parâmetro global do sistema que indica o número máximo de dimensões de uma tupla. A **chave** relacionada a uma tupla $t = \langle t_1, \ldots, t_l \rangle$, onde $l \leq D$, é calculada da seguinte forma: a tupla é normalizada para $D + 1$ dimensões na forma $t' = \langle l, t_1, \ldots, t_l, \perp_{l+1}, \ldots, \perp_D \rangle$. Na sequência, é calculado o **hash** (ex. SHA-1) de cada campo $t''_i = hash(t'_i)$, $\forall i \in [1, D + 1]$; a tupla $t'' = \langle t''_1, \ldots, t''_{D+1} \rangle$ é tratada como um ponto no espaço $D + 1$-dimensional e é mapeada em um índice pela função $k = Hilbert(t'')$. Esse procedimento é representado pela função $TupleKey(t)$. Para inserir a tupla t, calcula-se a chave $k = TupleKey(t)$ e insere-se a tupla no segmento responsável por k usando a camada de segmentação.

Note que é possível que tuplas distintas possuam a mesma chave, portanto $k \in \bar{K}$ não necessariamente implica que $t \leq_m \bar{t}$. Colisões de chaves de tuplas podem resultar de colisões dos **hashes** das coordenadas ou de truncamentos realizados pela função $Hilbert$, uma vez que esta recebe D parâmetros com 128 bits cada (se for usado SHA-1) e retorna um número do mesmo tamanho dos identificadores do **overlay**. Se o número de bits do identificador do **overlay** é menor que D multiplicado pelo número de bits da função **hash** utilizada, então alguma informação é perdida pela função $Hilbert$ de maneira que colisões podem ocorrer. No entanto, como a camada de segmentação permite a execução de requisições arbitrárias, podemos lidar com colisões enviando o próprio molde ao segmento responsável pelas chaves cobertas na busca. Dessa forma, as tuplas são encontradas localmente usando o molde e não somente a chave, logo apenas tuplas que efetivamente casam com o molde são retornadas. Assim, fica claro que a chave calculada para as tuplas não tem o papel de identificador, mas sim de estabelecer um local inequívoco para a tupla e permitir a busca eficiente. Como espaços de tuplas são baseados em busca por conteúdo, nenhuma semântica é perdida pela existência de colisões.

Para encontrar uma tupla a partir de um molde $\bar{t}$, procede-se de maneira similar ao procedimento de inserção. Primeiro o molde é normalizado para $D+1$ dimensões, depois é calculado o $hash$ de cada campo, porém em vez de calcular uma única chave com a função $Hilbert$, é necessário encontrar o conjunto de todas as chaves na curva unidimensional que correspondem às possíveis tuplas que casam com $\bar{t}$. Esse procedimento é representado pela função $TemplateKeys(\bar{t})$. Por não ser o escopo principal deste trabalho, não abordamos em detalhes o procedimento de computação das chaves usando a curva de Hilbert. Mais informações podem ser obtidas a partir de outros trabalhos da literatura que também utilizam o mesmo mecanismo de indexação para realizar buscas multidimensionais e por intervalos [Li and Parashar 2005, Lee et al. 2005, Shen et al. 2008].

O estado de um nó é dado por duas estruturas locais:

• ts: é um espaço de tuplas local que provê os métodos $ts.out(t)$, $ts.rdp(\bar{t})$ e $ts.inp(\bar{t})$ com a semântica usual. Além disso, essa estrutura contém um atributo $ts.limits$, um intervalo tal que toda tupla t com $TupleKey(t) \notin ts.limits$ é descartada. A operação de união de dois espaços de tuplas locais ($ts' \cup ts''$), com intervalos de limite consecutivos, é dada pela união de todas as tuplas contidas nos dois operandos e pela união dos intervalos $limits$. $limits$ corresponde ao intervalo de chaves do segmento definido na segmentação;

• $pending$: é um conjunto com entradas da forma $\langle C_i, \bar{t} \rangle$ que indicam que o nó i está aguardando uma tupla que case com o molde $\bar{t}$.

Para que o algoritmo de replicação garanta a consistência, é necessário que ts seja idêntico em todas as réplicas e retorne o mesmo resultado quando invocado com os mesmos parâmetros. Isso não é garantido pela semântica básica de um espaço de tuplas e implementações diferentes podem retornar resultados diferentes. Para evitar esse problema e garantir o determinismo das réplicas, pode-se assumir o uso de uma única implementação por todos os nós corretos do sistema ou pode-se aumentar a semântica do espaço de tuplas local com propriedades de ordenação sobre as tuplas. O importante é assegurar que, dado que i e j são réplicas do mesmo segmento, $ts_i.inp(\bar{t}) = ts_j.inp(\bar{t})$, mesmo se mais de uma tupla case com $\bar{t}$.

Algoritmo 1 Obtenção e alteração do estado local do Espaço de Tuplas

```
1:  upon SegGetAppState(start, end) do /* Upcall da segmentação para obter estado */
2:      T ← {t ∈ ts : TupleKey(t) ∈ [start, end)} /* Tuplas com chave dentro do intervalo */
3:      P ← {⟨C_i, t̄⟩ ∈ pending : TemplateKeys(t̄) ∪ [start, end) ≠ ∅} /* Pendencias que se sobrepõem ao intervalo */
4:      return ⟨T, P⟩
5:  upon SegSetAppState(start, end, ⟨⟨T^1, P^1⟩, ..., ⟨T^n, P^n⟩⟩) do /* Upcall da segmentação para alterar estado */
6:      ts.limits ← [start, end) /* Altera limites do espaço de tuplas local */
7:      ts ← ⋃_{i=1}^{n} T_i /* Guarda união das tuplas dos subestados */
8:      pending ← ⋃_{i=1}^{n} P_i /* Guarda união das pendências dos subestados */
```

O Algoritmo 1 descreve como o espaço de tuplas responde às chamadas da camada de segmentação para obter e alterar o estado. Na chamada $SegGetAppState$ são retornadas as tuplas dentro do intervalo especificado (linha 2) e as buscas pendentes nas quais as chaves do molde tem intersecção com o intervalo (linha 3). Na chamada $SegSetAppState$ são recebidos um intervalo de chaves e uma lista de estados parciais que precisam ser unidos. Os limites de ts são alterados para o intervalo passado (linha 6), as tuplas de todos os estados parciais são unidas e armazenadas em ts (linha 7) e as buscas pendentes também são reunidas (linha 8).

5.1. Inserção de Tupla

Algoritmo 2 Inserção de tuplas (out)

```
1:  operation out(t) /* Código do cliente p */
2:      k ← TupleKey(t) /* Gera chave da tupla */
3:      nonce ← GenerateNonce() /* Gera nonce para requisição */
4:      resp ←⊥ /* Busca e invoca segmento responsável pela chave da tupla */
5:      while resp =⊥ do
6:          SegFind(k, k)
7:          wait for SegFindOk(S_q)
8:          resp ← SegRequest(S_q, ⟨OUT, nonce, t⟩)
9:      upon SegDeliver(C_p, ⟨OUT, nonce, t⟩) do /* Código do servidor q */
10:         if TupleKey(t) ∈ ts.limits then /* Calcula chave da tupla e verifica se está nos limites do segmento */
11:             ts.out(t) /* Insere tupla no espaço local */
12:             SegResponse(C_p, ⟨OUT_OK⟩) /* Responde para cliente */
13:             for all ⟨C_i, t̄⟩ ∈ pending : t ≤_m t̄ do /* Notifica clientes pendentes interessados na nova tupla */
14:                 SegNotify(C_i, ⟨READ_OK, t⟩)
15:                 pending ← pending \ {⟨C_i, t̄⟩}
```

A inserção de tupla, dada pela operação $out(t)$ (Alg. 2), consiste inicialmente em calcular a chave relacionada à tupla t a ser inserida, por meio da função $TupleKey$ executada no cliente (linha 2). Na sequência, um laço de repetição é iniciado para buscar o segmento responsável pela tupla (linhas 6 e 7) e invocar o segmento passando a requisição para inserir a tupla (linha 8). Essa busca e inserção devem ser repetidas até que a resposta correta seja recebida. O uso de um **nonce** garante que a mesma requisição não é executada mais de uma vez.

No servidor, ao receber uma requisição de inserção (linha 9), a chave da tupla é verificada (linha 10) e caso a tupla pertença realmente ao segmento esta é inserida no

espaço de tuplas ts local (linha 11). Na sequência, o nó procura alguma busca pendente $\langle C_i, \bar{t} \rangle$ na qual a tupla inserida case com o molde $\bar{t}$ (linha 13). Para cada caso em que a tupla casar com o molde, uma notificação é enviada ao nó que efetuou a busca em questão (linha 14) e a pendência é removida (linha 15). O cliente notificado pode, então, prosseguir com a leitura ou exclusão. Note que a exclusão em si não é executada nesse ponto, ou seja, um nó registrado ao tentar excluir uma tupla, após receber a notificação, precisa ainda realizar uma requisição de exclusão. Mais detalhes sobre a exclusão de tuplas serão apresentados na Seção 5.4.

5.2. Busca de Tupla

As operações de leitura e exclusão de tuplas apresentam uma característica similar: calculam as chaves relacionadas ao molde de busca e varrem todos os segmentos responsáveis por essas chaves a fim de descobrir quais segmentos possuem tuplas que casem com um molde. Após encontrar os segmentos relevantes, as operações de leitura simplesmente retornam as tuplas encontradas, enquanto as operações de exclusão invocam o segmento para remover a tupla. Dada essa similaridade, decidimos escrever um procedimento genérico, que é utilizado pelas operações de leitura e exclusão, e que realiza a busca nos segmentos e notifica as tuplas que forem encontradas. Esse procedimento foi idealizado para ser executado em uma **thread** separada para que a busca de tuplas possa ocorrer de maneira concorrente.

A busca de uma tupla a partir de um molde consiste em calcular as chaves do molde, buscar todos os segmentos responsáveis por chaves contidas no conjunto de chaves e consultar os segmentos para ler uma tupla que case com o molde buscado. Se uma tupla adequada for encontrada durante a consulta, esta é notificada para a **thread** principal da operação. Por outro lado, se nenhuma tupla for encontrada, no caso bloqueante, o algoritmo aguarda o recebimento de notificações de segmentos sempre que uma tupla que casa com o molde for encontrada; no caso não bloqueante a busca termina imediatamente. No lado do servidor, ao receber um pedido de leitura, o nó busca seu espaço de tuplas local usando rdp e retorna o resultado, seja este uma tupla ou $\perp$, para o cliente. Se a leitura for bloqueante e não houver tupla adequada (resultado $\perp$), então o servidor registra o pedido na lista de pendências para que possa notificar o cliente posteriormente.

Note que a busca de tupla é similar seja bloqueante ou não, portanto decidimos fazer um algoritmo geral $FindTuple$ (Alg. 3) que engloba as duas funcionalidades e recebe um parâmetro $block$ que indica o caso específico. As operações específicas simplesmente invocam $FindTuple$ passando o molde e o parâmetro $block$ adequado. Cada tupla t encontrada no segmento S é notificada com a chamada $FindTupleOk(S, t)$. Essa notificação é tratada de maneira diferente pelas operações de leitura e de exclusão.

Há um ponto que adiciona complexidade ao algoritmo da busca de tuplas: a possibilidade de uma consulta a um segmento não ser executada efetivamente e retornar $\perp$ (i.e. segmento reconfigurou após ser descoberto). Para lidar com esses casos, é necessário construir o algoritmo com várias "passadas" pelas chaves remanescentes, cujos segmentos responsáveis ainda não puderam ser consultados, até que todo o conjunto seja efetivamente coberto. O tratamento do estado durante as reconfigurações (Alg. 1) garante que se um segmento com intervalo de chaves é consultado, o mesmo intervalo não precisa ser buscado novamente, mesmo que o segmento sofra reconfiguração. Tuplas e nós pendentes são propagados para segmentos novos garantindo a semântica das operações.

O procedimento $FindTuple$, no cliente (Alg. 3), inicia pelo cálculo do conjunto de chaves associado ao molde da busca (linha 2). Depois, o algoritmo entra em um **loop** (linhas 3 a 16) até que todas as chaves sejam efetivamente buscadas. Na primeira iteração, as chaves buscadas são exatamente as chaves do molde recém calculadas. À medida que segmentos forem consultados, o conjunto de chaves remanescentes é reduzido até que não reste nenhuma chave a ser buscada e o **loop** termina. No interior do **loop**, dois conjuntos são criados para controlar as chaves que foram buscadas e os segmentos consultados na passada atual, respectivamente $foundKeys$ (linha 4) e $doneKeys$ (linha 5). Na sequência, o conjunto de chaves remanescente é dividido em intervalos contíguos e para cada um destes, $SegFind$ é invocado a fim de encontrar os segmentos responsáveis (linha 7).

Algoritmo 3 Busca de tuplas ($FindTuple$)

```
 1: procedure FindTuple(t̄, block) /* Código do cliente p */
 2:     remainingKeys ← TemplateKeys(t̄) /* Calcula intervalos de chaves do molde */
 3:     while remainingKeys ≠ ∅ do /* Repete até que todas as chaves do molde sejam cobertas */
 4:         foundKeys ← ∅ /* Chaves que tiveram segmento encontrado */
 5:         doneKeys ← ∅ /* Chaves que tiveram segmento consultado com sucesso */
 6:         for all [k, k'] ∈ remainingKeys do /* Busca segmentos de todas as chaves pendentes */
 7:             SegFind(k, k')
 8:         while remainingKeys ⊈ foundKeys do /* Repete até todas as chaves pendentes terem segmento encontrado */
 9:             wait for SegFindOk(S_i) /* Segmento encontrado */
10:             foundKeys ← foundKeys ∪ K(S_i) /* Marca chaves do segmento encontrado */
11:             resp ← SegRequest(S_i, ⟨READ, t̄, block⟩) /* Consulta segmento */
12:             if resp = ⟨READ_OK, t⟩ then
13:                 doneKeys ← doneKeys ∪ K(S_i) /* Se consultou com sucesso, marca chaves do segmento */
14:                 if t ≠⊥ then /* Se encontrou tupla, realiza notificação */
15:                     FindTupleOk(S_i, t)
16:         remainingKeys ← remainingKeys \ doneKeys /* Não achou tupla na passada, remove chaves consultadas */
17:     if block then /* Todas as chaves do molde foram consultadas e nenhuma tupla encontrada. */
18:         wait for SegNotifyDeliver(S_i, ⟨READ_OK, t⟩) /* Busca bloqueante, aguarda notificação com tupla */
19:         FindTupleOk(S_i, t) /* Retorna tupla notificada */
20:     else
21:         return /* Busca não-bloqueante, termina sem aguardar notificação */
22: upon SegDeliver(C_p, ⟨READ, t̄, block⟩) do /* Código do servidor q */
23:     t ← ts.rdp(t̄) /* Busca tupla localmente */
24:     if block ∧ t =⊥ then /* Verifica necessidade de registrar pendência */
25:         pending ← pending ∪ {⟨C_p, t̄⟩} /* Cliente marcado como pendente */
26:     SegResponse(C_p, ⟨READ_OK, t⟩) /* Envia resposta contendo tupla ou ⊥ */
```

Depois de realizar $SegFind$ em todas as chaves, um outro laço de repetição (linhas 8 a 15) realiza o recebimento dos certificados de segmentos buscados na etapa anterior e a consulta dos segmentos correspondentes. Para cada segmento encontrado (linha 9), as chaves correspondentes são marcadas como encontradas (linha 10) e o segmento é invocado para buscar tuplas que casem com o molde (linha 11). Além do molde, a requisição contém também o parâmetro $block$, indicando se a busca é bloqueante ou não.

Depois de invocado o segmento, a resposta recebida é testada para saber se a invocação foi realizada de fato (linha 12). Se a resposta foi efetivamente recebida, as chaves do segmento são marcadas como consultadas (linha 13) e o valor recebido é testado (linha 14) para saber se uma tupla que casa com o molde foi encontrada. Se for o caso, o algoritmo notifica a **thread** principal sobre a tupla encontrada.

Depois que todos os segmentos foram invocados o conjunto $foundKeys$ passa a conter todas as chaves remanescentes e o laço interno termina. Então, todas as chaves de segmentos que foram efetivamente consultados são removidas das chaves remanescentes (linha 16). Se todas as chaves foram consultadas, então o laço de repetição externo termina. A partir desse ponto, se $block = true$, então os segmentos consultados terão sido

informados que o cliente deseja ser registrado como pendente (linha 11), o cliente aguarda alguma notificação vinda de um segmento e repassa qualquer tupla informada (linhas 18 - 19); se $block = false$, então o algoritmo termina.

No lado do servidor, os nós que recebem uma requisição de leitura realizam uma busca no ET local (linha 23). Caso a tupla não exista e o cliente esteja realizando uma busca bloqueante, o mesmo é registrado nas pendências (linha 25) para ser notificado quando uma tupla adequada for inserida, conforme a operação out (Alg. 2, linha 14). Por fim, a requisição é respondida com o resultado da busca local (linha 26).

5.3. Leitura de Tupla

As operações rd e rdp (Alg. 4) utilizam $FindTuple$ (Alg. 3) e retornam a primeira tupla encontrada. Assim, o código consiste em iniciar a **thread** de busca, aguardar uma notificação e retornar a tupla encontrada. Em todo caso, a operação termina ou quando uma tupla é retornada, ou quando o procedimento de busca termina. As condições de término do procedimento de busca são descritas na Seção 5.2.

Algoritmo 4 Leitura de tuplas bloqueante (rd) e não bloqueante (rdp)

```
1: operation rd(t̄) /* Código do cliente */
2:     FindTuple(t̄, TRUE) /* Inicia procedimento de busca de tupla */
3:     wait for FindTupleOk(S_i, t) /* Aguarda notificação de tupla encontrada */
4:     return t /* Retorna a tupla */
5: operation rdp(t̄) /* Código do cliente */
6:     FindTuple(t̄, FALSE) /* Inicia procedimento de busca de tupla */
7:     upon FindTupleOk(S_i, t) do /* Recebe notificação de tupla encontrada ou término da busca */
8:         return t /* Retorna a tupla */
```

5.4. Exclusão de Tupla

Na exclusão de tupla (in e inp), o espaço de chaves que corresponde ao molde é buscado para encontrar tuplas que casem. Porém, ao encontrar uma tupla, uma nova invocação é realizada para excluir a mesma. Se outro cliente remover a mesma tupla antes, então essa tentativa de exclusão falha. Separar a operação em duas fases, busca e exclusão, permite que executar a busca em paralelo para intervalos de chaves distintos. Já as invocações para exclusão são realizadas em série, uma vez que somente uma tupla deve ser excluída em cada chamada de in ou inp. Além disso, caso uma requisição de exclusão falhe (resposta $\perp$), esta precisa ser repetida até que um resultado seja obtido indicando se a tupla foi ou não removida, pois se a requisição foi executada e a resposta simplesmente atrasou, a operação deve parar e retornar a tupla já removida, evitando a remoção de outra.

As operações de exclusão bloqueante e não bloqueante são similares. Uma diferença é com relação ao tipo de busca, i.e., o parâmetro $block$ do procedimento $FindTuple$, que é $true$ para in e $false$ para inp. A outra diferença é com relação às tentativas de remover uma tupla encontrada: na operação bloqueante, quando o cliente tenta remover uma tupla que não está mais presente no espaço de tuplas, este deve ser registrado como pendente no segmento; na operação não bloqueante isso não é necessário. Sendo pequenas as diferenças, construímos um procedimento único $RemoveTuple$ que recebe um parâmetro $block$, como em $FindTuple$ (Seção 5.2). As operações in e inp (Alg. 6) simplesmente chamam $RemoveTuple$ passando o molde e o valor do parâmetro $block$ adequado.

O procedimento $RemoveTuple$ (Alg. 5) incia chamando $FindTuple$, passando o molde e o parâmetro $block$ (linha 2). Na sequência, ocorre o tratamento das tuplas que são encontradas durante a busca (linha 3). Primeiro um $nonce$ é gerado (linha 4) a fim de evitar remover mais de uma vez a mesma tupla e garantir idempotência. Depois, o segmento

responsável é invocado para remover a tupla (linha 5). Essa requisição inclui a tupla e o parâmetro $block$. Se a requisição falhar, um **loop** busca novamente o segmento responsável pela tupla (linha 7) e envia novamente a requisição de exclusão para o segmento encontrado (linha 9), garantindo que uma resposta efetiva seja recebida. Se essa resposta contiver a tupla e não o valor $\perp$, então a tupla é retornada e o procedimento termina.

Algoritmo 5 Exclusão de tuplas ($RemoveTuple$)

```
1:  procedure RemoveTuple(t̄, block) /* Código do cliente p */
2:      FindTuple(t̄, block) /* Inicia busca de tuplas. Quando (se) a busca termina, RemoveTuple também termina */
3:      upon FindTupleOk(S_i, t) do /* Tupla encontrada */
4:          nonce ← GenerateNonce() /* nonce evita que mesmo molde remova mais de uma tupla */
5:          resp ← SegRequest(S_i, ⟨REMOVE, t, block, nonce⟩) /* Envia requisição para segmento remover tupla */
6:          while resp = ⊥ do /* Repete até que resposta seja recebida */
7:              SegFind(TupleKey(t)) /* Busca novamente segmento responsável pela tupla */
8:              wait for SegFindOk(S_j)
9:              resp ← SegRequest(S_j, ⟨REMOVE, t, block, nonce⟩) /* Repete invocação com mesmo nonce */
10:         if resp = ⟨REMOVE_OK, t⟩ ∧ t ≠ ⊥ then
11:             return t /* Retorna tupla que foi removida */
12: upon SegDeliver(C_p, ⟨REMOVE, t, block, nonce⟩) do /* Código do servidor q */
13:     t ← ts.inp(t) /* Busca e exclui tupla localmente */
14:     if block ∧ t = ⊥ then /* Verifica necessidade de registrar pendência */
15:         pending ← pending ∪ {⟨C_p, t̄⟩} /* Cliente marcado como pendente */
16:     SegResponse(C_p, ⟨REMOVE_OK, t⟩) /* Envia resposta contendo a tupla ou ⊥ */
```

No lado servidor, os nós do segmento invocado para excluir uma tupla agem de maneira similar ao caso da busca de tupla. A diferença é que inp, em vez de rdp, é chamado no ET local, e uma tupla específica é buscada, em vez de um molde. Assim como na busca, se a tupla não é encontrada e se $block = true$, o cliente é registrado como pendente.

Algoritmo 6 Exclusão de tuplas bloqueante (in) e não bloqueante (inp)

```
1:  operation in(t̄) /* Código do cliente p */
2:      return RemoveTuple(t̄, TRUE) /* Inicia procedimento de remover tuplas bloqueante */
3:  operation inp(t̄) /* Código do cliente p */
4:      return RemoveTuple(t̄, FALSE) /* Inicia procedimento de remover tuplas não bloqueante */
```

6. Discussão sobre o Espaço de Tuplas

Os algoritmos apresentados na seção anterior implementam um ET tolerante a intrusões sobre um **overlay** P2P. A operação out (Seção 5.1) consiste simplesmente em calcular a chave da tupla usando a curva de Hilbert, depois buscar e invocar o segmento responsável pela chave calculada. Pelas propriedades de $SegFind$ e $SegRequest$, a busca e invocação podem precisar ser repetidas caso o segmento buscado reconfigure antes da invocação ou caso a resposta demore a chegar no nó cliente. Para o segundo caso, o algoritmo utiliza um **nonce** que evita a repetição da inserção, o que violaria a semântica da operação out.

As operações rd e rdp (Seção 5.3) consistem em uma varredura do conjunto de chaves representado pelo molde buscado, de forma que cada segmento responsável por chaves deste conjunto é consultado sobre a existência de uma tupla adequada. De acordo com as propriedades das curvas de preenchimento de espaço, quanto mais geral é o molde utilizado, maior é o conjunto de chaves coberto e maior é o número de segmentos consultados. Buscas por moldes mais gerais, portanto, implicam em um maior número de trocas de mensagens, porém a consulta a diferentes segmentos é realizada em paralelo, reduzindo o tempo de resposta geral das operações rd e rdp. Assim como na inserção de tupla, é preciso repetir a busca e consulta a segmentos sempre que o segmento reconfigurar antes da invocação. No caso de rd (bloqueante), caso a tupla não seja encontrada o cliente aguarda notificações vindas dos segmentos consultados.

As operações in e inp (Seção 5.4) iniciam de maneira similar às operações de leitura. No caso de in (bloqueante), a operação também aguarda o recebimento de notificações posteriores caso a tupla não seja encontrada. Caso mais de uma tupla seja encontrada concomitantemente, é importante que as invocações de exclusão sejam realizadas em série, evitando que uma única operação in ou inp termine na exclusão de mais de uma tupla. Como a exclusão da tupla é uma invocação destrutiva, então um **nonce** também é utilizado pelo mesmo motivo, como na inserção.

A operação out realiza a verificação das buscas pendentes e notifica todos os nós bloqueados com moldes que casem com a tupla inserida. Todo nó notificado é excluído do conjunto de pendências. Para todos os nós bloqueados em operações rd, a simples notificação de uma tupla basta para encerrar o bloqueio. Por outro lado, no caso de nós bloqueados em operações in, o nó precisa ainda tentar remover a tupla para terminar a operação e caso a mesma tupla acabe sendo excluída antes, o nó precisa continuar bloqueado. Por esse motivo, a invocação para excluir a tupla pode fazer com que o nó entre novamente no conjunto de pendências. Uma característica interessante desse mecanismo de notificação é que se múltiplos nós bloqueados aceitam uma mesma tupla, a finalização de todas as leituras é garantida, mesmo que haja uma ou mais exclusões pendentes.

7. Trabalhos Relacionados

Linda [Gelernter 1985] é uma linguagem de coordenação que introduziu o conceito de ET e comunicação generativa. Diversas implementações de ET foram desenvolvidas e incluídas em plataformas de coordenação, como JavaSpaces (`http://www.jini.org/wiki/JavaSpaces\_Specification`) e IBM TSpaces (`http://www.almaden.ibm.com/cs/TSpaces/`). Essas soluções armazenam as tuplas em um servidor centralizado, limitando a escalabilidade e a tolerância a falhas.

Algumas soluções usam replicação para garantir disponibilidade e confiabilidade do ET [Xu and Liskov 1989, Hansen and Cannon 1994, Bakken and Schlichting 1995, Bessani et al. 2008]. O DepSpace [Bessani et al. 2008] é um ET tolerante a faltas bizantinas que possui propriedades de segurança como confidencialidade e controle de acesso.

Lime [Picco et al. 1999] é um ET para redes móveis, onde cada nó possui seu ET local. Uma visão distribuída do ET é construída a partir do agrupamento do ET local com o ET de nós vizinhos na rede. Apesar da distribuição, o sistema não tolera saídas ou falhas, uma vez que as tuplas locais não são replicadas.

Comet [Li and Parashar 2005] é uma implementação de espaços de tuplas sobre uma DHT **Chord** que utiliza curvas de Hilbert para distribuir as tuplas pelos nós do **overlay**. No entanto, o mesmo não utiliza replicação e não tolera faltas.

8. Conclusão

Este artigo apresentou uma construção de espaço de tuplas sobre a infraestrutura de segmentação proposta em [Böger et al. 2012]. O espaço de tuplas foi elaborado na forma de algoritmos distribuídos que realizam as operações seguindo a semântica usual. Uma análise informal dos algoritmos levantou aspectos específicos da implementação, incluindo limitações e formas de contorná-las. Além de servir como demonstração da utilidade da segmentação, os algoritmos representam uma contribuição por si mesmos, uma vez que, conforme apresentado nos trabalhos relacionados, há poucas iniciativas para usar espaços de tuplas em P2P e nenhum trabalho nesse contexto somando-se tolerância a intrusões.

Referências

Bakken, D. and Schlichting, R. (1995). Supporting fault-tolerant parallel programming in linda. *Par. and Dist. Syst., IEEE Trans. on*, 6(3):287 –302.

Baldoni, R., Querzoni, L., Virgillito, A., Jimenez-Peris, R., and Virgillito, A. (2005). Dynamic quorums for dht-based p2p networks. In *Net. Comp. and App., 4th IEEE Int. Symp. on*, pages 91–100.

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

Böger, D. S., Fraga, J., Alchieri, E., and Wangham, M. (2012). Intrusion-tolerant shared memory through a p2p overlay segmentation. In *Adv. Inf. Net. and App. (AINA), 2012 IEEE 26th Int. Conf. on*, pages 779 –786, Fukuoka, JP. IEEE.

Castro, M., Druschel, P., Ganesh, A., Rowstron, A., and Wallach, D. (2002). Secure routing for structured peer-to-peer overlay networks. *SIGOPS Oper. Syst. Rev.*, 36(SI):299–314.

Dwork, C., Lynch, N., and Stockmeyer, L. (1988). Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323.

Gelernter, D. (1985). Generative communication in linda. *ACM Trans. Prog. Lang. Syst.*, 7(1):80–112.

Hansen, R. and Cannon, S. (1994). An efficient fault-tolerant tuple space. In *Fault-Tol. Par. and Dist. Syst., 1994., Proc. of IEEE Work. on*, pages 220 –225.

Lamport, L., Malkhi, D., and Zhou, L. (2010). Reconfiguring a state machine. *SIGACT News*, 41(1):63–73.

Lamport, L., Shostak, R., and Pease, M. (1982). The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401.

Lawder, J. and King, P. (2000). Using space-filling curves for multi-dimensional indexing. In Lings, B. and Jeffery, K., editors, *Adv. in Dat.*, volume 1832 of *LNCS*, pages 20–35. Springer Berlin / Heidelberg.

Lee, J., Lee, H., Kang, S., Choe, S., and Song, J. (2005). Ciss: An efficient object clustering framework for dht-based peer-to-peer applications. In Ng, W., Ooi, B.-C., Ouksel, A., and Sartori, C., editors, *Dat., Inf. Syst., and Peer-to-Peer Comp.*, volume 3367 of *LNCS*, pages 215–229. Springer Berlin Heidelberg.

Li, Z. and Parashar, M. (2005). Comet: a scalable coordination space for decentralized distributed environments. In *Hot Topics in Peer-to-Peer Systems, 2005. HOT-P2P 2005. 2nd Int. Work. on*, pages 104 – 111.

Picco, G., Murphy, A., and Roman, G.-C. (1999). Lime: Linda meets mobility. In *Soft. Eng., 1999. Proc. of the 1999 Int. Conf. on*, pages 368 –377.

Rowstron, A. and Druschel, P. (2001). Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In Guerraoui, R., editor, *Middleware 2001*, volume 2218 of *LNCS*, pages 329–350. Springer, Berlin.

Shen, D., Shao, Y., Nie, T., Kou, Y., Wang, Z., and Yu, G. (2008). Hilbertchord: A p2p framework for service resources management. In Wu, S., Yang, L., and Xu, T., editors, *Adv. in Grid and Perv. Comp.*, volume 5036 of *LNCS*, pages 331–342. Springer Berlin Heidelberg.

Wallach, D. (2003). A survey of peer-to-peer security issues. In Okada, M., Pierce, B., Scedrov, A., Tokuda, H., and Yonezawa, A., editors, *Soft. Sec. - Theo. and Syst.*, volume 2609 of *LNCS*, pages 253–258. Springer Berlin / Heidelberg.

Xu, A. and Liskov, B. (1989). A design for a fault-tolerant, distributed implementation of linda. In *Fault-Tol. Comp., 19th Int. Symp. on*, pages 199 –206.