Segmentação de Overlays P2P como Suporte para Memórias Tolerantes a Intrusões

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

[1]Departamento de Automação e Sistemas –UFSC
Florianópolis – SC – Brasil

[2]Grupo de Sistemas Embarcados e Distribuídos – UNIVALI
São José – SC – Brasil

{dsboger,fraga,alchieri}@das.ufsc.br, wangham@univali.br

Abstract. *This paper describes our experience in developing an infrastructure which allows building intrusion-tolerant shared memory for large-scale systems. The infrastructure makes use of a P2P overlay and of the concept of State Machine Replication (SMR). Segmentation is introduced on the key space of the overlay to allow the use of algorithms for supporting SMR. In this paper we describe the proposed infrastructure in its stratification and corresponding algorithms. Also, analyses of the algorithms described and their respective costs are presented.*

Resumo. *Este artigo descreve nossa experiência no desenvolvimento de uma infraestrutura que permite a construção de memórias compartilhadas tolerantes a intrusões para sistemas de larga escala. A infraestrutura faz uso de um overlay P2P e do conceito de Replicação Máquina de Estados (RME). O conceito de segmentação é introduzido sobre o espaço de chaves do* overlay *para permitir o uso de algoritmos de suporte à RME. No presente artigo descrevemos a infraestrutura proposta em sua estratificação e algoritmos. Além disso, realizamos uma análise da solução apresentada e dos custos envolvidos.*

1. Introdução

As redes par a par (*peer-to-peer*, P2P) correspondem a um paradigma de comunicação que tem sido o foco de muitos trabalhos nos últimos anos. O uso desse paradigma foi bastante popularizado na Internet, principalmente por ser a base para as aplicações de compartilhamento de arquivos modernas. Diversas outras aplicações já foram desenvolvidas usando P2P, como *multicast* e sistemas de e-mail [Steinmetz and Wehrle 2005]. Apesar disso, P2P ainda é pouco utilizado em aplicações mais complexas que poderiam se beneficiar de aspectos como a escalabilidade [Baldoni et al. 2005].

As principais características que tornam as redes P2P uma arquitetura interessante para sistemas distribuídos são o uso eficiente dos recursos ociosos disponíveis na Internet e a capacidade de aumento do número de nós sem detrimento do desempenho. Normalmente as redes P2P oferecem primitivas de comunicação com latência e número de mensagens de ordem logarítmica em relação ao número de nós presentes na rede [Stoica et al. 2001, Rowstron and Druschel 2001].

Apesar das suas vantagens, as redes P2P apresentam desafios para o provimento de garantias de confiabilidade. Essas redes normalmente são formadas dinamicamente por nós totalmente autônomos que podem entrar e sair do sistema a qualquer momento. Essas características de dinamismo tornam difícil manter a consistência das informações distribuídas no sistema. Ademais, essas redes não possuem uma gerência global, sendo redes de pares normalmente abertas. Devido a essa característica, as redes P2P podem conter participantes maliciosos que colocam em risco o funcionamento das aplicações.

Neste trabalho, é apresentada uma infraestrutura tolerante a intrusões para memórias compartilhadas em sistemas de larga escala. Esta infraestrutura faz uso das funcionalidades de um *overlay* P2P estruturado e tolerante a intrusões, sobre o qual é introduzido o conceito de segmentação tomando como base a divisão do espaço de chaves da rede P2P e a aplicação de técnicas de Replicação Máquina de Estados (RME) [Schneider 1990]. A partir de segmentos é possível garantir consistência e tolerar uma quantidade de participantes maliciosos em um sistema de memória compartilhada. A segmentação do *overlay* depende do fornecimento de uma técnica de indexação, isto é, um mapeamento de objetos para chaves, pela aplicação de memória compartilhada.

O restante do artigo está organizado da seguinte forma: a seção 2 enumera as premissas do modelo de sistema adotado, a seção 3 introduz a solução proposta neste trabalho, a seção 4 contém discussões sobre as propostas de nosso trabalho e coloca problemas em aberto. A seção 5 examina os trabalhos relacionados e conforta os mesmos diante de nossas soluções. A seção 6 traz as conclusões do artigo.

2. Modelo de Sistema

Consideramos um sistema distribuído formado por um conjunto finito Π de processos (ou nós) extraídos de um universo U, interconectados por 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 é considerado correto se age de acordo com a especificação dos protocolos nos quais participa. Um nó malicioso (ou bizantino [Lamport et al. 1982]) pode agir de maneira arbitrária ou simplesmente parar em certos momentos. O sistema proposto tolera certo número de nós maliciosos durante sua execução. Assume-se que em qualquer momento da execução, no máximo f nós faltosos estão presentes no sistema. O parâmetro f é global e conhecido por todos os nós do sistema.

Imediatamente acima da rede, encontram-se duas camadas independentes que serão usadas para construir a camada de segmentação proposta neste trabalho. A camada de *overlay*, descrita na Seção 2.1, implementa uma rede P2P sobre o sistema, com busca eficiente de nós distribuídos, e a camada de suporte à replicação, descrita na Seção 2.2, que fornece uma abstração de Replicação Máquina de Estados (RME) [Schneider 1990] usada para garantir a disponibilidade e consistência das informações contidas no sistema. Em geral, os custos da RME não permitem que essa técnica seja aplicada a uma grande quantidade de nós, portanto neste trabalho dividimos o sistema

Tabela 1: Camadas do Sistema

Segmentação	
Overlay	**Suporte à Replicação**
Rede	

em diversas máquinas de estados independentes. A **camada de segmentação** faz uso dessas duas e provê meios para invocar eficientemente qualquer RME do sistema. A Tabela 1 apresenta as camadas do sistema.

A camada de rede é acessada a partir de duas operações: A operação $Send(q, m)$ envia a mensagem m para o nó de endereço q. A operação $Receive(m)$ aguarda o recebimento de uma mensagem qualquer m. Os canais de comunicação da rede são ponto a ponto e confiáveis, ou seja, não há perda ou alteração de mensagens. 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 serão usados nas camadas superiores. No entanto, não há garantia de sincronismo por toda a execução.

2.1.Camada de *Overlay*

Acima da camada de rede, assume-se a existência de um *overlay* que provê operações similares às redes P2P estruturadas, como *Pastry* [Rowstron e Druschel 2001] e *Choord* [Stoica et al. 2003]. Essas redes atribuem identificadores aos nós e distribuem estes em torno de um anel lógico. Nós conhecem outros nós com identificadores numericamente próximos, denominados de vizinhos, de forma a manter a estrutura do anel. Além disso, os nós possuem tabelas de roteamento que são usadas para contatar eficientemente nós distantes no anel. Aplicações se utilizam dessa estrutura definindo esquemas para a distribuição de objetos pelo *overlay*, normalmente atribuindo chaves aos objetos e armazenando esses objetos em nós com identificadores numericamente próximos a essas chaves. A busca por um objeto consiste em usar as tabelas de roteamento para encontrar nós com identificador próximo à chave associada ao mesmo.

Este trabalho utiliza o *overlay* tolerante a faltas bizantinas definido por Castro et al. (2002), o qual apresenta a propriedade de garantir alta probabilidade na entrega de mensagens a todos nós corretos na vizinhança de uma chave, mesmo se uma quantidade de nós do *overlay* for maliciosa. A confiabilidade da entrega é alcançada pelo envio de uma mensagem por múltiplas rotas e pelo uso de tabelas de roteamento especiais que aumentam a probabilidade de que essas rotas sejam disjuntas, ou seja, não tenham nós em comum. O número de rotas disjuntas, ao superar o limite estabelecido de faltas garante a entrega das mensagens. Análises e resultados experimentais [Castro et al. 2002] demonstram que a probabilidade de entrega de uma mensagem é de 99,9% se a proporção de nós maliciosos for de até 30%.

O funcionamento do *overlay* depende da geração e atribuição segura de identificadores a nós da rede, de forma que nós maliciosos não possam escolher seu identificador nem participar do *overlay* com múltiplas identidades. Essas propriedades são garantidas, por exemplo, pelo uso de certificados que associam o identificador do nó a seu endereço de rede e sua chave pública. Esses certificados são emitidos por uma autoridade certificadora (AC) confiável, que também pode ser responsável por gerar identificadores aleatórios para os nós[1]. Na nossa infraestrutura, mantemos o uso desses certificados na camada de segmentação, onde é denominado de **certificados de nó**.

[1] Não é necessário que esta AC seja uma PKI oficial. A mesma pode ser uma comissão de gestão do sistema, um administrador, etc. O identificador pode ser gerado a partir de uma função *hash* aplicada ao endereço de rede do nó.

Nas seções a seguir, é assumido o uso do *overlay* definido por Castro et al. (2002), no entanto, outras construções P2P similares podem ser utilizadas sem alteração das camadas superiores da nossa proposta. O *overlay* deve suportar, então, para entrada e saída de um nó p, operações $OverlayJoin(C_p)$ e $OverlayLeave(C_p)$ que são concretizadas através da apresentação de seu certificado C_p; $OverlaySend(k, m)$ para enviar uma mensagem m para os nós vizinhos da chave k; e $OverlayDeliver(m)$ que entrega a mensagem m para a camada superior nos nós de destino.

2.2. Camada de Suporte à Replicação

A camada de replicação, que implementa protocolos para Replicação Máquinas de Estados (RME) [Schneider 1990], é usada pelas camadas superiores para prover garantias de disponibilidade e consistência das informações mesmo em presença de nós faltosos e maliciosos. Em ambientes com atividades intrusivas, MEs são mantidas através do uso de algoritmos de consenso tolerantes a faltas bizantinas (p. ex. [Castro e Liskov 1999]). No presente texto não é definido um algoritmo específico de suporte à RME. Qualquer um pode ser escolhido, desde que tolere f nós faltosos de um total de no mínimo n_{MIN} ($n_{MIN} \geq 3f + 1$ [Lamport et al. 1982]).

A camada de replicação oferece às camadas superiores as operações: $TOMulticast(P, m)$ e $TODeliver(m)$. A primeira operação garante o envio de uma mensagem m aos processos do conjunto P e a segunda define a entrega de mensagens segundo ordenação total aos processos de P. As duas operações caracterizam, portanto, um *multicast* com ordenação total (*total order multicast*).

Como as redes P2P são dinâmicas, assume-se o uso de algoritmos com capacidade de reconfiguração, ou seja, algoritmos que permitam a modificação na composição de processos integrantes da RME. Lamport et al. (2008) definem formas simples para transformar um modelo de replicação estático em um dinâmico. No presente trabalho, é assumida a operação $TOReconfigure(P')$, que altera o parâmetro local da ME que indica o conjunto de processos. A chamada $TOReconfigureOk(P')$ notifica a camada superior o momento em que a chamada $TOReconfigure(P')$ acaba.

3. Solução Proposta

A Tabela 2 apresenta as camadas e operações especificadas na Seção 2. Sobre o *overlay* e a replicação, é definida uma camada de segmentação, descrita na Seção 3.1, na qual os nós do *overlay* são distribuídos em um conjunto de segmentos. Cada segmento executa uma RME distinta e é responsável por um conjunto de chaves do *overlay*.

Tabela 2: Camadas e Primitivas

Segmentação: $SegJoin(C_p)$, $SegLeave(C_p)$, $SegFind(k, k')$, $SegFindOk(S)$, $SegRequest(S_q, req)$, $SegDeliver(C_p, req)$, $SegResponse(C_p, resp)$, $SegGetAppState()$, $SegSetAppState(start, end, state)$	
Overlay: $OverlayJoin(C_p)$, $OverlayLeave(C_p)$, $OverlaySend(k, m)$, $OverlayDeliver(m)$	**Suporte à Replicação:** $TOMulticast(P, m)$, $TODeliver(m)$, $TOReconfigure(P')$, $TOReconfigureOk(P')$
Rede: $Send(q, m)$, $Receive(m)$	

3.1. Camada de Segmentação

A camada de segmentação divide o anel lógico do *overlay* em **segmentos** compostos de nós contíguos, no qual cada segmento é 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 ME 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. A cada reconfiguração, um novo conjunto de nós (denominado **configuração** ou **visão**) passa a compor o segmento e a executar os algoritmos associados de suporte à RME. O número de nós de um segmento pode aumentar ou diminuir, logo para evitar que segmentos fiquem com número de nós abaixo do limite de n_{MIN} requerido pelos algoritmos de RME, pode ser necessário unir dois segmentos contíguos em um segmento maior. Por outro lado, o aumento do número de nós de um segmento para um valor muito superior a n_{MIN} pode comprometer o desempenho dos algoritmos de RME. Para aliviar o problema, segmentos grandes podem ser divididos em segmentos menores. É importante notar que reconfigurações ocorrem localmente nos segmentos e não no sistema inteiro. O número máximo de nós em um segmento é um parâmetro global do sistema denotado n_{MAX}. Quando o segmento atinge o número de n_{MAX} nós, é preciso dividir os membros em dois segmentos vizinhos, logo n_{MAX} deve ser maior ou igual a $2n_{MIN}$. Além disso, é necessário adicionar uma tolerância ao valor de n_{MAX}, caso contrário, uma única saída (ou entrada) após uma divisão (ou união) de segmentos dispararia uma união (ou divisão). Esse fato pode ser explorado por nós maliciosos para provocar sucessivas uniões e divisões, deteriorando o desempenho da RME.

Segmentos são descritos por **certificados de segmento** (S), que contém a lista de todos os certificados de nó (C_p) dos membros do segmento e um contador de configurações ($confId$) incrementado a cada reconfiguração da ME. Quando ocorre uma reconfiguração, os membros do segmento antigo decidem o novo conjunto de membros e assinam um novo certificado de segmento (ou dois, em caso de divisão do segmento). Se ocorrer uma união de segmentos, membros dos dois segmentos devem assinar o certificado do novo segmento. Assim, cria-se uma cadeia de assinaturas que pode ser usada para verificar a validade de um certificado atual a partir de um segmento inicial aceito globalmente (possivelmente assinado por um administrador confiável).

Tabela 3: Estruturas de Dados da Camada de Segmentação

$C_p = \langle addr, pubK, id \rangle \sigma_{CA}$	é o certificado do nó p, no qual $addr$ é o endereço de rede do mesmo, $pubK$ e id são, respectivamente, a chave pública e o identificador no *overlay* de p. Esse certificado é o mesmo utilizado na camada de *overlay*
$privK_p$	é a chave privada do nó p correspondente à chave pública presente em C_p e é usada em assinaturas digitais pelo nó[1]. Assume-se que mensagens recebidas com assinaturas inválidas não são processadas por nós corretos
$S_p = \langle members, confId, start, end, \Sigma, history \rangle$	é o certificado do segmento atual de um nó p, no qual $members$ é o conjunto dos certificados de nó dos membros atuais do segmento, $confId$ é o contador de configurações, $[start, end) = K(S_p)$ é o intervalo de chaves do *overlay* pelas quais o segmento é responsável, Σ é um conjunto de assinaturas e $history$ é a cadeia de certificados representando o caminho de evolução do segmento atual
S_p^+ e S_p^-	são certificados de segmentos vizinhos do segmento de S_p, representando, respectivamente, o seu sucessor e seu antecessor no anel de segmentos. Ambos apresentam a mesma estrutura de S_p
$changes$	conjunto de alterações na lista de membros a serem aplicadas na próxima reconfiguração. A entrada do nó i é denotada por $\langle +, i \rangle$ e a saída por $\langle -, i \rangle$
$reconfigCount$	contador de nós que solicitam reconfiguração na configuração atual

```
1.  operation SegFind(k, k')
2.      /* Código do cliente p */
3.          keys ← ∅ /* conjunto de chaves cobertas pelos certificados recebidos */
4.          OverlaySend(k, ⟨FIND, C_p, S_p, k, k'⟩σ_p) /* envia mensagem assinada usando o overlay */
5.          while [k, k'] \ keys ≠ ∅ do /* teste de cobertura completa */
6.              /* aguarda respostas dos servidores */
7.              wait for Receive(⟨FIND_OK, C_q, S_q⟩σ_q) /* recebe resposta de algum servidor q */
8.              if K(S_q) ∩ keys ≠ ∅ ∧ ValidHistory(S_q) then /* testa segmento */
9.                  keys ← keys ∪ K(S_q) /* atualiza cobertura de chaves */
10.                 SegFindOk(S_q) /* notifica que segmento foi encontrado */
11.             end if
12.         end while

13.     /* Código do servidor q */
14.         upon OverlayDeliver(⟨FIND, C_p, S_p, k, k'⟩σ_p) do
15.             if [k, k'] ∩ K(S_q) ≠ ∅ then
16.                 Send(C_p.addr, ⟨FIND_OK, C_q, S_q⟩σ_q) /* envia resposta assinada */
17.                 if k' ≥ S_q.end then /* segmento não cobre espaço de chaves; repassar requisição */
18.                     OverlaySend(S_q.end, ⟨FIND, C_p, S_p, k, k'⟩σ_p)
19.                 end if
20.             end if
21. end operation
```

Algoritmo 1: Busca de Segmentos

Cada segmento executa uma RME que é responsável por parte do estado da aplicação. Para invocar uma requisição em uma máquina de estados, um cliente deve primeiro encontrar o segmento responsável pela máquina (usando a camada de *overlay*) e depois enviar a requisição para a máquina (usando a camada de suporte à replicação). A Tabela 3 apresenta as estruturas de dados e as seções a seguir apresentam os algoritmos da camada de segmentação.

3.1.1. Busca e Invocação de Segmentos

Para invocar uma RME, um nó precisa primeiro obter o certificado do segmento que executa essa máquina de estados. A busca de segmentos é realizada pela operação $SegFind$ (Algoritmo 1), que tem como parâmetro de entrada um intervalo de chaves e retorna um conjunto de certificados dos segmentos responsáveis pelas chaves nesse intervalo. A operação encontra certificados fazendo primeiro uma busca no *overlay* pela primeira chave do intervalo (linha 4). Os nós que receberem a mensagem de busca pelo *overlay* responderão enviando seus certificados de segmento (linha 16) e repassando a mesma para segmentos vizinhos caso o intervalo de chaves buscado se estenda além do seu próprio segmento (linhas 17 a 19). A cada certificado de segmento S recebido pelo cliente, a operação chama a função $ValidHistory(S)$, que verifica se a cadeia de segmentos que acompanha S é válida. Se o encadeamento e as assinaturas forem válidos, a operação invoca $SegFindOk(S)$ para notificar a camada superior (linhas 7 a 11). A operação no cliente termina quando todo o intervalo de chaves buscado for coberto pelos certificados de segmento recebidos (teste da linha 5).

De posse de um certificado de segmento, o nó pode executar a operação $SegRequest$ (Algoritmo 2) fazendo uso do suporte à RME. Essa operação consiste em iniciar um temporizador, invocar o segmento usando a $TOMulticast$ (linha 6), passando o $confId$ do certificado que o cliente conhece (linha 4), e aguardar $f + 1$

respostas idênticas de membros distintos (linhas 7 a 13). Se o certificado conhecido pelo cliente for antigo, os servidores não repassarão a requisição para a aplicação e o cliente não receberá $f + 1$ respostas, o que provocará o estouro do temporizador a operação irá retornar $\perp$, indicando uma exceção (linhas 14 e 15). A operação $SegRequest$ não trata das exceções e deixa essa responsabilidade para as camadas superiores. Se o certificado de segmento usado na invocação for atual, a requisição é entregue para a camada superior pela *upcall* $SegDeliver$ (linha 19). As respostas a requisições são enviadas pela operação $SegResponse$ por meio de mensagens de resposta endereçadas ao cliente (linha 24).

3.1.2. Entrada e Saída de Nós

A entrada do nó p na camada de segmentos se dá pela operação $SegJoin(C_p)$ (Algoritmo 3), na qual C_p é o certificado de nó de p. Antes de entrar em algum segmento, p invoca $OverlayJoin(C_p)$ e entra no *overlay* (linha 3). Depois, p busca o certificado do segmento responsável por seu identificador (linha 6) e invoca o mesmo passando uma mensagem $JOIN$ (linhas 7 e 8). Após obter uma resposta válida, p aguarda o recebimento dos certificados de segmento e do estado da aplicação (linhas 11 a 22), enviados pelos membros do segmento. O estado é repassado à camada superior e a operação $TOReconfigure$ é chamada (linha 23), alterando os nós participantes da RME para o novo segmento de p. Quando os membros do segmento recebem a mensagem $JOIN$ do nó p (linha 25), simplesmente registram o pedido de p no conjunto $changes$ (linha 26) e respondem (linha 27). A reconfiguração ocorre posteriormente, na

```
1.  operation SegRequest(S_q, req)
2.      /* Código do cliente p */
3.          StartTimer( ) /* inicia temporizador */
4.          knownId ← S_q.confId /* identificador de configuração conhecido por p */
5.          resps ← ∅ /* conjunto de respostas a receber */
6.          TOMulticast(S_q.members, ⟨REQ, C_p, knownId, req⟩σ_p) /* requisição com ordenação total */
7.          upon Receive(⟨RESP, C_q, resp_q⟩σ_q) do /* recebimento de uma resposta */
8.              if C_q ∈ S_q.members then
9.                  resps ← resps ∪ resp_q
10.                 if #_{resp_q} resps = f + 1 then
11.                     return resp_q /* retorna a resposta */
12.                 end if
13.             end if
14.         upon Timeout( ) do
15.             return ⊥ /* ocorreu um estouro de temporizador */

16.     /* Código do servidor q */
17.         upon TODeliver(⟨REQ, C_p, knonwnId, req⟩σ_p) do
18.             if knownId = S_q.confId then
19.                 SegDeliver(C_p, req)  /* requisição é entregue para aplicação */
20.             end if
21. end operation

22. operation SegResponse(C_p, resp_q)
23.     /* Código do servidor q */
24.         Send(C_q.addr, ⟨RESP, C_q, resp_q⟩σ_q)
25. end operation
```

Algoritmo 2: Invocação de Segmentos

operação $SegReconfigure$, conforme descrito no Algoritmo 4.

A saída de nós é realizada pela operação $SegLeave(C_p)$ (Algoritmo 3). O nó envia uma mensagem $LEAVE$ (linha 31) para os membros do seu segmento e continua a atender requisições até o término da próxima reconfiguração, notificada pela operação $TOReconfigureOk$ (linha 32). Depois o nó termina de atender requisições e invoca $OverlayLeave(C_p)$ (linha 33). Nós corretos são obrigados a aguardar a reconfiguração para garantir o funcionamento dos algoritmos de replicação. De maneira similar à operação de entrada, os nós que recebem a mensagem $LEAVE$ registram o pedido de p (linha 36) e a reconfiguração propriamente dita se dá pela operação $SegReconfigure$.

```
1.  operation SegJoin(C_p)
2.      /* Código do cliente p */
3.          OverlayJoin(C_p) /* entrada no overlay */
4.          resp ←⊥
5.          while resp =⊥ do /* tenta até achar um segmento atual que responda diferente de ⊥ */
6.              SegFind(C_p.id, C_p.id) /* busca pelo segmento responsável pelo identificador de p */
7.              wait for SegFindOk(S_q)
8.              resp ← SegRequest(S_q, ⟨JOIN⟩) /* invocação do segmento em que p entrará /
9.          end while
10.         /* transferência do estado da aplicação */
11.         states ← ∅ /* cópias do estado que serão recebidas */
12.         stateReceived ← FALSE
13.         while not stateReceived do /* repete até receber f + 1 cópias iguais do estado */
14.             wait for Receive(⟨STATE, C_q, ⟨S_i, S_i^+, S_i^−, appState_i⟩⟩ σ_i) /* Enviado pelos membros do
    segmento antigo (Algoritmo 4, linhas 36 a 39) */
15.             state_i ← ⟨S_i, S_i^+, S_i^−, appState_i⟩
16.             states ← states ∪ state_i
17.             if #_{state_i} states = f + 1 then /* recebidas f + 1 cópias iguais do estado */
18.                 S_p ← S_i; S_p^+ ← S_i^+; S_p^− ← S_i^− /* guarda certificados de segmento */
19.                 SegSetAppState(S_p.start, S_p.end, appState_i) /* repassa para camada superior */
20.                 stateReceived ← TRUE
21.             end if
22.         end while
23.         TOReconfigure(S_p.members) /* configura replicação da máquina de estados */

24.     /* Código do servidor q */
25.         upon SegDeliver(C_p, ⟨JOIN⟩) do
26.             change ← changes ∪ {⟨+, C_p⟩} /* registra alteração da lista de membros */
27.             SegResponse(C_p, ⟨JOIN_OK⟩)
28. end operation

29. operation SegLeave(C_p)
30.     /* Código do cliente p */
31.         SegRequest(S_p, ⟨LEAVE⟩) /* invoca o próprio segmento */
32.         wait for TOReconfigureOk(S) /* reconfiguração que exclui p terminou */
33.         OverlayLeave(C_p) /* sai do overlay */

34.     /* Código do servidor q*/
35.         upon SegDeliver(C_p, ⟨LEAVE⟩) do
36.             changes ← changes ∪ {⟨−, C_p⟩} /* registra alteração da lista de membros */
37.             SegResponse(C_p, ⟨LEAVE_OK⟩)
38. end operation
```

Algoritmo 3: Entrada e saída de nós em segmentos

3.1.3. Reconfiguração de Segmentos

A reconfiguração de um segmento ocorre quando $f + 1$ nós executam a operação $SegReconfigure$ (Algoritmo 4). Essa operação é invocada após certo tempo, indicado pelo estouro do temporizador $ReconfigTimeout$. Nesse momento, o nó assina e dissemina no segmento uma mensagem de tentativa de reconfiguração (linhas 3 a 5). Essas tentativas de reconfiguração (recepções de mensagens $TRY\_RECONFIG$) são acumuladas pelos nós do segmento (linha 8) e quando algum nó recebe $f + 1$ dessas tentativas, dispara uma requisição ordenada para a reconfiguração, enviando juntamente as tentativas assinadas como prova (linhas 9 a 11). Como as tentativas não são ordenadas, é possível que a requisição de reconfiguração seja invocada mais de uma vez, porém o teste da linha 15 garante que a reconfiguração de fato somente ocorrerá uma vez por segmento. Além disso, a reconfiguração é ordenada juntamente com os pedidos de entrada e saída, o que garante que todos os nós corretos possuem o mesmo conjunto $changes$ e, portanto, calcularão o mesmo conjunto de membros.

O código da reconfiguração consiste inicialmente em calcular o novo conjunto de membros (linha 17) e checar o tamanho do conjunto de novos membros a fim de detectar se será necessário dividir ou unir segmentos, de acordo com o tamanho do conjunto e com os parâmetros globais n_{MIN} e n_{MAX} (linhas 18 e 20). Se não for o caso, ocorrerá uma reconfiguração simples (linhas 23 a 39), que consiste em gerar e assinar localmente um novo certificado de segmento (linhas 23 e 24), disseminar e coletar as assinaturas de $f + 1$ membros do segmento antigo (linhas 25 a 32) e montar o novo certificado com as assinaturas coletadas (linha 34). Depois, o estado da aplicação local é obtido pela chamada $SegGetAppState$ e disseminado para os novos membros (linhas 35 a 38) e o algoritmo de RME é reconfigurado com $TOReconfigure$ (linha 39).

3.1.4. Divisão e União de Segmentos

A divisão de segmentos ocorre quando o número de nós em um segmento excede uma constante n_{MAX}. Essa constante é conhecida globalmente e indica um número de nós a partir do qual é aconselhável formar duas MEs. A divisão em si consiste inicialmente em dividir o conjunto total de membros em dois subconjuntos de maneira que um subconjunto $S^{<}$ conterá metade dos nós com menor identificador e o outro subconjunto $S^{>}$ a outra metade com identificador superior. Cada segmento será responsável por um intervalo de chaves definido da seguinte forma: seja $[k, k')$ o intervalo de chaves do segmento antigo e p o membro que possui o menor identificador do subconjunto $S^{>}$, então $K(S^{<}) = [k, C_p.id)$ e $K(S^{>}) = [C_p.id, k')$. O $confId$ dos novos certificados será o sucessor do $confId$ do segmento atual. A assinatura e transferência do estado são similares à reconfiguração simples, porém são dois certificados assinados e para os novos membros são enviados apenas o estado referente ao segmento do qual farão parte. A reconfiguração da RME ocorre de maneira similar à reconfiguração simples.

A união de segmentos é um pouco mais complexa que a divisão, pois a reconfiguração envolve duas máquinas de estados em execução. A união é iniciada quando o número de nós do segmento ficará menor que os n_{MIN} necessários para manter as propriedades da RME. Os nós do segmento que iniciou a união (pelo menos $f + 1$ corretos) enviam mensagens simples para os nós do seu segmento sucessor a fim de notificar a necessidade de uma união. Essa mensagem deve conter o novo conjunto de membros do segmento, conforme calculado na operação $SegReconfigure$, para que

o segmento sucessor possa calcular os membros do novo segmento resultante da união. De maneira similar à operação $SegReconfigure$, quando um nó do segmento vizinho recebe $f + 1$ pedidos de união válidos, este invoca uma operação na RME do próprio segmento para que uma reconfiguração de união ocorra. Os nós do segmento vizinho executam uma operação similar à divisão, no sentido de gerar um novo certificado de segmento. Este novo segmento conterá todos os membros dos dois segmentos, terá $confId$ superior aos valores nos dois segmentos envolvidos na união e terá intervalo de

```
1.  operation SegReconfigure( )
2.      upon ReconfigTimeout( ) do
3.          for all C_i ∈ S_p.members do
4.              Send(C_i.addr, ⟨TRY_RECONF, C_p, S_p.confId⟩σ_p) /* tentativa de reconfiguração */
5.          end for
6.      upon Receive(⟨TRY_RECONF, C_i, confId_i⟩σ_i) do
7.          if confId_i = S_q.confId then /* testa se tentativa não está atrasada (não ordenada) */
8.              reconfigCount ← reconfigCount ∪ {⟨TRY_RECONF, C_i, confId_i⟩σ_i}
9.              if #reconfigCount ≥ f + 1 then /* número mínimo de tentativas alcançado */
10.                 TOMulticast(S_q.members, ⟨RECONFIG, C_p, reconfigCount⟩σ_q) /* realiza
reconfiguração com chamada ordenada */
11.             end if
12.         end if

13. /* Código do servidor q */
14.     upon TODeliver(⟨RECONFIG, C_p, reconfigCount_p⟩σ_p) do
15.         if #reconfigCount_p ≥ f + 1 ∧ ∀r ∈ reconfigCount_p : r.confId = S_q.confId ∧
              ValidSig(r) then /* tentativas suficientes, assinaturas válidas, relativas ao seg. atual */
16.             /* calcula novo conjunto de membros */
17.             newMembers ← (S_q.members ∪ {C_i : ⟨+, C_i⟩ ∈ changes}) \ {C_i : ⟨−, C_i⟩ ∈ changes}
18.             if #newMembers < n_MIN then /* necessário unir segmentos */
19.                 Merge(newMembers)
20.             else if #newMembers > n_MAX then /* necessário dividir segmento */
21.                 Split(newMembers)
22.             else /* reconfiguração simples */
23.                 newConfId ← S_q.confId + 1
24.                 σ_q ← Sign(⟨newMembers, newConfId, S_q.start, S_q.end⟩)
25.                 newΣ ← ∅ /* disseminação da assinatura */
26.                 for all C_i ∈ S_q.members do Send(C_i.addr, ⟨NEW_SIG, C_q, σ_q⟩) end for
27.                 while #newΣ < f + 1 do
28.                     wait for Receive(⟨NEW_SIG, C_i, σ_i⟩)
29.                     if ValidSig(σ_i, ⟨newMembers, newConfId, S_q.start, S_q.end⟩) then
30.                         newΣ ← newΣ ∪ {σ_i}
31.                     end if
32.                 end while
33.                 newHistory ← S_q.history ∪ {S_q} /* certificado atual entra no histórico */
34.                 S_q ← ⟨newMembers, newConfId, S_q.start, S_q.end, newΣ, newHistory⟩
35.                 appState_q ← SegGetAppState( ) /* upcall para obter estado da aplicação */
36.                 for all C_i : ⟨+, C_i⟩ ∈ changes do
37.                     Send(C_i.addr, ⟨STATE, C_q, S_q, appState_q⟩σ_q)
38.                 end for
39.                 TOReconfigure(S_q.members) /* reconfigura máquina de estados */
40.             end if
41.             changes ← ∅ /* limpa registro de entradas e saídas */
42.             reconfigCount ← ∅ /* reinicia contador de pedidos de reconfiguração */
43.         end if
44. end operation
```

Algoritmo 4: Reconfiguração de Segmento

chaves igual à união dos dois intervalos. Os membros dos dois segmentos trocam os estados da aplicação para formar um estado agregado e depois disso enviam o estado agregado aos novos membros (que estavam registrados nos conjuntos $changes$ dos dois segmentos). A reconfiguração da RME ocorre como na reconfiguração simples.

4. Considerações sobre a Infraestrutura Proposta

A camada de segmentação tem a função de permitir o uso eficiente de algoritmos de suporte à RME (Replicação Máquina de Estado) em ambientes de larga escala. Normalmente esses algoritmos possuem custo quadrático em função do número de participantes, o que torna pouco escalável e bastante custosa sua aplicação direta em redes P2P. Com a solução proposta neste trabalho, é possível prover confiabilidade e tolerância a intrusões para aplicações executando sobre redes P2P de grande porte.

A operação $SegFind$ consiste de uma simples busca no *overlay* por um intervalo de chaves. O Algoritmo 1 descreve uma busca recursiva, na qual segmentos adicionais são buscados apenas depois que o segmento anterior é encontrado. Para grandes intervalos de chaves, é possível dividir os mesmos e realizar buscas em paralelo nos correspondentes subintervalos, porém um número grande de buscas paralelas pode levar a redundância, isto é, múltiplas buscas podem chegar ao mesmo segmento. A terminação da operação está ligada à garantia de entrega de mensagens provida pela camada de *overlay*. Como o *overlay* utilizado é probabilístico, existe uma pequena chance da operação $SegFind$ ficar bloqueada aguardando respostas porque o *overlay* falhou. Uma solução simples seria usar um temporizador que levaria à repetição da busca.

Outro aspecto importante da operação $SegFind$ é a verificação, por meio do histórico de segmentos, da validade dos certificados de segmento recebidos na busca. Essa verificação pode implicar em um alto custo, uma vez que o histórico é um conjunto de segmentos que aumenta à medida que o sistema evolui. Uma otimização que diminui o custo de processamento consiste em manter em cada nó um cache de certificados válidos. Toda vez que um certificado é validado, por meio da verificação das assinaturas, este é adicionado ao cache. Validações subsequentes do mesmo certificado não precisariam verificar as assinaturas, uma vez que este estaria presente no cache. Essa otimização tem um efeito importante, pois quanto mais antigo é um segmento, maior a probabilidade de ser compartilhado por vários históricos de segmentos mais atuais. Para reduzir o custo de transmissão, ao realizar uma busca, um nó pode enviar na requisição certificados de segmento contidos no cache local que interseccionam com o intervalo de chaves procurado. Caso os certificados enviados façam parte do histórico dos segmentos requisitados, os nós que responderem a requisição podem podar os históricos de certificados que atendem a demanda. Ou seja, os históricos não precisam ser enviados completos para as validações. São excluídos dos históricos todos os certificados anteriores aos certificados enviados na requisição de busca.

A operação $SegRequest$ apresenta apenas o custo de uma invocação da RME, uma vez que o certificado de segmento contém os endereços dos nós correspondentes. Pode ocorrer, no entanto, que o certificado usado no momento da invocação esteja ultrapassado por conta de uma reconfiguração no segmento invocado. Nesse caso, os nós não responderão à requisição, e faz-se o uso de um temporizador para retirar o cliente da espera. Pode ocorrer de o temporizador estourar por causa de um atraso na

Tabela 4: Custo típico das operações

Operação	Mensagens	Rodadas
$SegFind$	$O(f \log N + N_{Seg}^2)$	$\log N + 3 + \left\lceil \frac{k'-k}{K_{Seg}} \right\rceil$
$SegRequest$	$O(N_{Seg}^2)$	5
$SegJoin$	$O(f \log N + N_{Seg}^2)$	$\log N + 8 + \left\lceil \frac{k'-k}{K_{Seg}} \right\rceil$
$SegLeave$	$O(N_{Seg}^2)$	5
$SegReconfigure$	$O(N_{Seg}^2)$	8 (reconf. simples e divisão), 13 (união)

rede e não por conta da reconfiguração do segmento e a repetição da invocação provocará uma execução dupla da requisição. Cabe à aplicação buscar novamente o certificado com $SegFind$ e garantir a idempotência das operações, por exemplo, com uso de *nonces*. Para garantir que a requisição da aplicação será efetivamente respondida, é preciso manter a premissa de que o número total de reconfigurações do sistema é finito, ou pelo menos que o número de reconfigurações concorrentes com uma requisição é finito. Essa premissa é usada em outros sistemas dinâmicos descritos na literatura para garantir terminação [Aguilera et al. 2009].

A operação $SegJoin$ faz uso das operações $SegFind$ e $SegRequest$ em um laço de repetição como seria usado na camada de aplicação. Dessa forma, também depende da premissa descrita acima para terminar. A operação $SegLeave$ é direcionada diretamente ao próprio segmento do nó cliente, ou seja, não é possível que o certificado de segmento seja antigo em nós corretos. Uma limitação dessa operação é que nós corretos precisam aguardar a próxima reconfiguração para saírem do sistema. Isso é necessário para que as requisições à ME, bem como a assinatura do próximo certificado e a transferência do estado da aplicação possam sempre ocorrer corretamente.

A operação $SegReconfigure$ é a que apresenta o maior custo de toda a camada de segmentação por conta da necessidade de união e divisão de segmentos. No caso de uma reconfiguração simples (sem divisão nem união) há uma rodada de disseminação de assinaturas e depois a transferência de estado para os segmentos novos. No caso de divisão de segmento são duas assinaturas, porém o custo de mensagens é o mesmo e a transferência de estado é igualmente simples. A união é muito mais custosa, uma vez que envolve a invocação de outro segmento e a transferência de estado ocorre também entre os membros dos segmentos antigos antes desse estado ser enviado aos novos membros. Além disso, se muitos nós entrarem ou saírem dos segmentos próximos ao mesmo tempo, pode ser necessário realizar mais de uma divisão ou união em sequência.

A Tabela 4 apresenta aproximações dos custos de mensagens (assintóticas) e de rodadas relativos às operações da camada de segmentação em situações típicas, sem levar em conta as otimizações descritas nesta seção. Os valores são derivados dos algoritmos da Seção 3.1 e assumem que $N_{Seg} = O(f)$ é o número médio de nós por segmento, K_{Seg} é o tamanho médio do intervalo de chaves dos segmentos, o custo da invocação na RME é $O(N_{Seg}^2)$ mensagens e demanda 5 rodadas [Castro e Liskov 1999], o custo esperado de um envio usando o *overlay* é $O(f \log N + f^2)$ mensagens e $\log N + 3$ rodadas, onde N é o número de nós no sistema [Castro et al. 2002].

5. Trabalhos Relacionados

Rosebud [Rodrigues e Liskov 2003] é um sistema de armazenamento distribuído que apresenta características similares a um *overlay* P2P estruturado. Para garantir disponibilidade e consistência diante de nós maliciosos, dados são replicados em $3f + 1$ nós e quóruns de $2f + 1$ nós são usados para realizar leitura e escrita. O sistema permite entrada e saída de nós por meio de um esquema de reconfiguração. Diferentemente da proposta deste trabalho, a reconfiguração é controlada por um subconjunto de nós do sistema, chamado de serviço de reconfiguração. A visão completa do sistema é mantida pelo serviço de reconfiguração e certificados contendo essa visão são disseminados a todos os nós a cada reconfiguração. No nosso trabalho, evitamos a necessidade de conhecimento completo do sistema, com o intuito de aliviar problemas de escala e também para mantermos a nossa proposta mais de acordo com a filosofia P2P que enfatiza a inexistência de gerenciamentos globais.

Castro et al. (2002) apresentam a proposta de um *overlay* P2P que garante alta probabilidade na entrega de mensagens mesmo quando uma parcela dos nós do sistema é maliciosa. A garantia de entrega é uma propriedade importante em redes P2P e permite a construção de soluções mais completas para prover tolerância a falhas e intrusões em redes P2P, como a que apresentamos neste trabalho. No próprio artigo, Castro et al. (2002) descrevem brevemente uma solução para leitura e escrita consistente de objetos mutáveis em uma rede P2P usando o roteamento confiável juntamente com técnicas de RME. Há certa similaridade com a solução proposta neste trabalho, porém aqui estendemos a replicação para suportar quaisquer operações, e tratamos de questões como entrada e saída de nós, além de união e divisão de segmentos.

Bhattacharjee et al. (2007) definem uma solução de roteamento P2P tolerante a intrusões com base na divisão do sistema em segmentos dinâmicos que podem sofrer divisões ou uniões à medida que o sistema evolui. Porém, estes segmentos são em geral maiores dos que aqueles adotados neste trabalho e não estão associados a RMEs. Cada segmento possui um subconjunto de nós, o comitê, que participam de uma RME e têm a função de decidir sobre as reconfigurações do segmento, disseminando os certificados de novos segmentos. A confiabilidade das informações emitidas pelo comitê é garantida pelo uso de criptografia de limiar assimétrica que permite a recriação do segredo compartilhado em caso de reconfiguração do comitê. Neste trabalho evitamos a criação de uma hierarquia, pois isso adicionaria complexidade e custo no gerenciamento dos segmentos que seriam formados por dois tipos de nós. Além disso, adotamos o mecanismo de encadeamento de certificados de segmento como meio de verificação dos mesmos para evitar os altos custos de reconfiguração da criptografia de limiar (obtenção de novas chaves parciais em cada atualização dos segmentos).

6. Conclusão

Este trabalho apresentou uma infraestrutura para a construção de memórias distribuídas de larga escala com base em um overlay P2P tolerante a intrusões. Esta infraestrutura utiliza segmentação para permitir o uso eficiente de técnicas de Replicação Máquina de Estados. Aspectos de dinamismo do sistema como entradas e saídas de nós, além de união e divisão de segmentos, foram também descritos no texto. Considerações sobre os custos e as limitações da infraestrutura foram levantadas e algumas otimizações foram

citadas. Para demonstrar a utilidade da infraestrutura proposta, um espaço de tuplas foi desenvolvido sobre a mesma. Os próximos passos deste trabalho compreendem a formalização das provas de funcionamento dos algoritmos propostos e a obtenção de resultados experimentais por meio de simulações e testes.

8. Referências

Aguilera, M. K., Keidar, I., Malkhi, D. e Shraer, A. (2009) "Dynamic Atomic Storage Without Consensus", In: *Proceedings of the PODC'09*, pp. 17-25, ACM.

Baldoni, R., Jiménez-Peris, R., Patiño-Martinez, M. e Virgillito, A. (2005) "Dynamic Quorums for DHT-based P2P Networks", In: *Proceedings of the NCA'05*, IEEE.

Bhattacharjee, B., Rodrigues, R. e Kouznetsov, P. (2007) "Secure Lookup without (Constrained) Flooding", In: *Proceedings of the WRAITS'07*, pp. 13-17.

Castro, M., Liskov, B. (1999) "Practical Byzantine Fault Tolerance", In: *Proceedings of the OSDI'99*, USENIX.

Castro, M., Druschel, P., Ganesh, A., Rowstron, A. e Wallach, D. S. (2002) "Secure Routing for Structured Peer-to-Peer Overlay Networks", In: *Proceedings of the OSDI'02*, USENIX.

Dwork, C., Lynch, N. e Stockmeyer, L. (1988) "Consensus in the Presence of Partial Synchrony", In: *Journal of the ACM*, v. 35, n. 2, pp. 288-323, ACM.

Gelernter, D. (1985) "Generative Communication in Linda", In: *ACM Transactions on Programming Languages and Systems*, v.7, n. 1, pp. 80-112, ACM.

Lamport, L., Shostak, R., Pease, M. (1982) "The Byzantine generals problem". *ACM TOPLAS*, v. 4, n. 3, pp. 382-401, ACM.

Rodrigues, R. e Liskov, B. (2003) "Rosebud: A Scalable Byzantine-Fault-Tolerant Storage Architecture", Relatório Técnico, MIT.

Rowstron, A. I. T., Druschel, P. (2001) "Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems", In: *Proceedings of the Middleware'01*, Springer.

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

Steinmetz, R. e Wehrle, K. (Eds.) (2005) *Peer-to-Peer Systems and Applications*, LNCS, v. 3485, Springer.

Stoica, I., Morris, R., Liben-Nowell, D., Karger, D. R., Kaashoek, M. F., Dabek, F. e Blakrishnan, H. (2003) "Chord: Scalable Peer-to-peer Lookup Protocol for Internet Applications", In: *ACM/IEEE Transactions on Networking (TON)*, v. 11, n. 1, IEEE Press.