VCube-PB: Uma Solução Publish/Subscribe Autonômica e Escalável com Difusão Confiável Causal

João Paulo de Araujo[1], Luiz A. Rodrigues[2], Luciana Arantes[1], Elias P. Duarte Jr.[3] e Pierre Sens[1]

[1] Sorbonne Universités, UPMC, CNRS, Inria, França

[2]Colegiado de Ciência da Computação – Univ. Estadual do Oeste do Paraná, Brasil

[3]Departamento de Informática – Univ. Federal do Paraná, Brasil

joao.araujo@lip6.fr, luiz.rodrigues@unioeste.br, luciana.arantes@lip6.fr, elias@inf.ufpr.br, pierre.sens@lip6.fr

Abstract. *This paper presents VCube-PB, a topic-based Publish/Subscribe (Pub/Sub) solution implemented on top of VCube, a virtual hypercube-like topology that organizes the processes based on their states (faulty or fault-free). Any published message regarding a topic is sent to subscribers through a spanning tree built on the hypercube with root in the publisher of the message. Messages are propagated using reliable broadcast and delivered to the subscribers respecting the causal order. Experiments with different scenarios were conducted on top of the simulator PeerSim and the results confirm the scalability of the proposed solution, as well as its good performance in dynamic scenarios.*

Resumo. *Este trabalho apresenta VCube-PB, uma solução de Publish/Subscribe (Pub/Sub) baseada em tópicos implementada sobre o VCube, uma topologia virtual baseada em hipercubo que organiza os processos do sistema com base nos seus estados (falho ou sem falha). Toda mensagem publicada referente a um tópico é enviada aos assinantes (subscribers) interessados neste tópico por meio de uma árvore geradora hierárquica criada dinamicamente sobre o hipercubo com raiz no nodo editor (publisher) da mensagem. A difusão é confiável e o sistema de Pub/Sub garante a entrega das mensagens respeitando a precedência causal entre elas. Experimentos com o simulador PeerSim foram realizados com diferentes cenários e os resultados confirmam a escalabilidade da solução proposta, bem como seu bom desempenho em cenários com dinamicidade.*

1. Introdução

Sistemas *Publish/Subscribe* (*Pub/Sub*) são caracterizados por uma rede de nodos distribuídos em que um ou mais nodos editores (*publishers*) produzem mensagens (eventos) e nodos assinantes (*subscribers)*, as consomem [Baldoni et al. 2005, Esposito et al. 2013]. A comunicação entre editores e assinantes é coordenada pelos próprios nodos, garantindo a entrega das mensagens produzidas a todos os assinantes nelas interessados de forma assíncrona.

Existem basicamente dois modelos de sistemas de *Pub/Sub*: os baseados em tópicos (*topic-based*) e os baseados em conteúdo (*content-based*) [Eugster et al. 2003]. No modelo em tópicos, as mensagens são classificadas em assuntos pré-definidos. Exemplos

de tópicos seriam "SBRC 2017" e "Previsão do tempo em Belém". Um assinante pode se cadastrar em quantos tópicos quiser e irá receber todas as mensagens relacionadas aos tópicos em que se registrou. Já no modelo baseado em conteúdo, os eventos são estruturados na forma de atributos múltiplos. O assinante deve, portanto, explicitar em quais atributos e valores ele tem interesse. Um exemplo seria: "temperatura: $< 30\,^{\circ}\mathrm{C}$","local = Belém", "mês = maio". A vantagem do primeiro modelo é que a classificação das mensagens em tópicos é estática, a difusão de mensagens aos assinantes é geralmente baseada em grupos *multicast* e, a interface oferecida ao usuário é simples. Por outro lado, o modelo em conteúdo oferece uma maior flexibilidade ao assinante para poder definir os eventos que lhe interessam, porém uma interface de usuário mais complexa.

Neste trabalho, estamos interessados em sistemas de *Pub/Sub* por tópicos e, particularmente, em oferecer uma forma eficiente, confiável e escalável para a difusão de mensagens entre assinantes de um tópico e a garantia de entrega destas mensagens respeitando a ordem causal entre elas.

No sistema proposto, denominado *VCube-PB*, um assinante registra ou remove a assinatura para o tópico t invocando as funções SUBSCRIBE(t) ou UNSUBSCRIBE(t), respectivamente. Um nodo publica uma mensagem m associada ao tópico t, invocando a função PUBLISH(t, m). Inspirado em Rodrigues et al. (2014), a mensagem m é transmitida de forma confiável (*reliable broadcast*) a todos os assinantes não falhos de t por meio de uma árvore geradora (*spanning tree*), cuja raiz é o nodo editor de m. Esta árvore é construída dinamicamente pelo módulo denominado *VCube-Top*. Trata-se de uma extensão do sistema de diagnóstico *VCube* [Duarte et al. 2014], que organiza todos os nodos do sistema em uma topologia virtual em hipercubo e, a cada rodada de testes de monitoração, informa os nodos da composição do sistema, considerando que nodos podem falhar. O *VCube* apresenta importantes propriedades logarítmicas, mesmo quando processos falham. No caso do *VCube-Top*, existem múltiplos hipercubos lógicos pois os nodos são logicamente organizados em um hipercubo por tópico, não necessariamente completo. Estes se reorganizam automaticamente quando há novos registros ou cancelamentos de assinaturas para o tópico em questão ou quando falhas de nodos são detectadas. Observe que o sistema é considerado autonômico, pois nodos se monitoram e automaticamente se adaptam às mudanças. Enfatizamos também que as modificações na composição dos assinantes de um tópico são transmitidas aos demais assinantes e editores somente a cada nova rodada de testes do *VCube-Top*. Além disso, a precedência causal entre mensagens relacionadas a um mesmo tópico é respeitada. Para implementar tal ordem causal, utilizamos o princípio de "barreira causal" (*causal barrier*) [Raynal 2013], em função da sua escalabilidade.

Vale ressaltar que existem sistemas de *Pub/Sub* na literatura baseados em árvores de difusão [Castro et al. 2006, Zhuang et al. 2001, Gao et al. 2011]. No entanto, a árvore gerada por estes sistemas é fixa, a sua manutenção por vezes custosa e, toda difusão de uma nova publicação é realizada por um único nodo raiz que, consequentemente, pode se tornar um gargalo para o bom desempenho do sistema. No *VCube-PB*, árvores geradoras são criadas dinamicamente a cada publicação de mensagem, não tendo custo de manutenção mesmo em caso de falhas ou mudança na composição dos assinantes/editores e tendo como raiz não um mesmo nodo fixo, mas o nodo editor da mensagem. Um outro ponto a assinalar é que poucos sistemas de *Pub/Sub* existentes na literatura (por exemplo, JEDI [Cugola et al. 2001]) garantem a precedência causal entre mensagens.

O restante do texto está organizado nas seguintes seções. A Seção 2 descreve trabalhos relacionados. A Seção 3 define o modelo do sistema e a interface em termos das funções. Discutimos, na Seção 4, a difusão causal confiável de publicações num contexto dinâmico de assinaturas. Na Seção 5, os sistemas *VCube* e *VCube-Top* são apresentados, assim como os algoritmos propostos para o *VCube-PB*. A Seção 6 contém os resultados da avaliação experimental, enquanto a Seção 7 conclui o trabalho.

2. Trabalhos Relacionados

Scribe [Castro et al. 2006] é um sistema de *Pub/Sub* baseado em tópicos em que toda nova assinatura é roteada por meio do *overlay peer-to-peer* Pastry [Rowstron e Druschel 2001] até encontrar um outro assinante, criando assim uma árvore geradora. Todos os eventos são propagados pela árvore a partir da raiz do tópico, que é fixa. De forma similar, Bayeux [Zhuang et al. 2001] utiliza o *overlay* Tapestry para implementar o serviço de inscrição e disseminação de mensagens a partir do nodo editor. Marshmallow [Gao et al. 2011] é uma solução baseada em conteúdo que também utiliza um modelo de inscrição agrupado em árvores sobre o *overlay* Pastry. Quando um nodo deseja publicar uma mensagem, ele a encaminha para o nodo raiz responsável pelo atributo correspondente, que a propaga por meio da árvore *multicast*.

TIBCO Rendezvous [TIBCO 2016] utiliza uma abordagem distribuída em uma arquitetura *peer-to-peer*. No caso de rede local, mensagens são propagadas por meio de *broadcast* UDP e as mensagens são filtradas por tópico localmente em cada nodo assinante. Gryphon [Strom et al. 1998], Siena [Carzaniga et al. 2001], JEDI [Cugola et al. 2001] e Hermes [Pietzuch e Bacon 2002] utilizam uma abordagem híbrida, com múltiplos servidores (*brokers*) distribuídos, responsáveis por garantir os requisitos da aplicação (consistência, confiabilidade, disponibilidade, etc.) e filtragem/roteamento das mensagens. Por exemplo, no JEDI, a disseminação de eventos/mensagens é feita por meio de uma árvore de servidores e clientes podem se conectar em qualquer nodo da árvore. A ordenação causal dos eventos é garantida. No entanto, essa arquitetura estática sobrecarrega os servidores raiz e causa interrupção do serviço em caso de falha dos servidores intermediários.

Google Pub/Sub [Google 2016] é um serviço baseado em tópicos que visa à alta disponibilidade e escalabilidade utilizando múltiplos servidores de forma balanceada. A solução é utilizada por diversos aplicativos da empresa, como *Ads* e *Gmail*, mas também pode ser contratado por terceiros. GraPS [Canas et al. 2015] utiliza um modelo baseado em grafos. Pontos de interesse são mapeados em vértices e as arestas representam o relacionamento entre pontos.

Outras soluções de *Pub/Sub* se encontram nos *surveys* [Baldoni et al. 2005], [Eugster et al. 2003], [Hinze e Buchmann 2010] e [Esposito et al. 2013].

3. Modelo do Sistema e Funções Oferecidas

Considera-se um sistema distribuído composto por um conjunto finito $\Pi = \{p_0, .., p_{n-1}\}$ com $n = 2^d$ processos, $d > 0$, que se comunicam por troca de mensagens. Cada processo está alocado em um nodo distinto. Assim, os termos *nodo* e *processo* são utilizados com o mesmo sentido.

A rede é representada por um grafo completo com enlaces confiáveis: cada par de processos está conectado diretamente por enlaces ponto-a-ponto bidirecionais e mensagens trocadas entre dois processos nunca são perdidas, corrompidas ou duplicadas. O

sistema admite falhas por parada (*crash*) e estas são permanentes. Um processo que nunca falha é denominado *correto*. Caso contrário, ele é *falho*. O sistema é considerado síncrono, ou seja, os atrasos máximos para a velocidade de execução de processos e transmissão de mensagens são conhecidos.

Funções disponibilizadas à aplicação: um nodo registra e remove seu interesse em um determinado tópico t, isto é, se torna ou deixa de ser assinante de t, invocando as funções SUBSCRIBE(t) e UNSUBSCRIBE(t) respectivamente. Um nodo registra e remove seu interesse em publicar mensagens para um tópico t invocando as funções PUBLISHER_REGISTER(t) e PUBLISHER_UNREGISTER(t) respectivamente. Estas funções são necessárias para que o nodo editor receba ou cesse de receber as modificações referentes à composição do grupo dos assinantes de t. Tendo se registrado como editor para o tópico t, um nodo pode publicar uma mensagem m invocando a função PUBLISH(t, m). Note que um nodo editor de um tópico pode ser ou não assinante deste tópico, o que caracteriza um grupo *multicast* aberto [Raynal 2013].

4. Difusão Confiável Baseada em Tópicos com Ordenação Causal

Sem perda de generalidade e por questões de legibilidade dos algoritmos, definiu-se a função TOPIC_TAG (t), aplicada a cada tópico. Essa função define um valor exclusivo de identificação do tópico t dentro do intervalo [1, .., MAX_TOPICS]. O valor de "*topic_tag*" é incluído nas mensagens do sistema *Pub/Sub*.

No sistema *VCube-PB*, a publicação de uma mensagem relacionada a um tópico é feita por meio de um algoritmo de difusão que garante, **por tópico**, uma difusão causal confiável (*reliable causal broadcast*). Um nodo correto que assina o tópico t é considerado um **assinante correto** após a primeira entrega de uma mensagem de t até o momento em que ele remove seu interesse por t ou falha. As seguintes propriedades garantem a confiabilidade do algoritmo de difusão:

- **Validade** (*validity*): se um processo correto i publica por meio de difusão uma mensagem m, i também entregará m em um tempo finito;
- **Integridade** (*integrity*): toda mensagem m é entregue por um processo no máximo uma vez (não-duplicação) e somente se a difusão de m foi realizada anteriormente (não-criação);
- **Acordo** (*agreement*): se um processo correto entrega a mensagem m referente ao tópico t, então todo assinante correto de t entrega m em um tempo finito.

Por uma questão de coerência, o algoritmo de difusão garante também a ordem causal entre as mensagens publicadas para um mesmo tópico. Assim, se um processo correto entrega m e depois publica m', todo processo correto, assinante de t, entrega m' somente depois de ter entregue m. No entanto, nodos entram e saem dinamicamente do sistema *Pub/Sub* proposto, pois um nodo não é necessariamente um assinante de um tópico desde o início da execução do sistema e, uma vez assinado em um ou mais tópicos, pode cancelar as inscrições. Portanto, a causalidade entre a difusão de duas mensagens somente pode ser satisfeita para um dado assinante, se este último já estivesse no sistema quando da difusão da primeira mensagem.

Ordem causal: se um processo publica uma mensagem m' após uma mensagem m ter sido entregue para ele, nenhum outro processo entregará m depois de m'.

Para implementar a causalidade no algoritmo de difusão do sistema *Pub/Sub*, utilizaremos o princípio de *barreira causal* [Raynal 2013]. A vantagem desta abordagem

é que o controle da causalidade não é baseado na identificação dos nodos, mas sim na dependência direta de mensagens, o que torna o algoritmo adequado à dinamicidade dos nodos (assinaturas, cancelamento de assinaturas ou falha de nodos).

Sejam m e m' duas mensagens da aplicação. A mensagem m **precede diretamente** a mensagem m' (denotado por $m \prec_{im} m'$) se a difusão de m precede causalmente a difusão de m' e não existe m'' tal que a difusão de m preceda causalmente a difusão de m'' e esta preceda causalmente a difusão de m'.

A barreira causal de m (cb_m) consiste no conjunto de mensagens que precedem diretamente m. A Figura 1 retrata a difusão de mensagens em uma rede em que, inicialmente, três nodos (p_0, p_1 e p_2) estão presentes e são assinantes corretos de um tópico t. À esquerda está uma visão do envio e recebimento das mensagens ao longo do tempo e, à direita, o grafo com as dependências entre as mensagens. No exemplo, a entrega de m_3 está condicionada à entrega anterior de m_1 ($m_1 \prec_{im} m_3$), visto que p_1 recebeu m_1 antes de iniciar a difusão de m_3 ($cb_{m_3} = \{m_1\}$). Já m_4 só pode ser entregue após a entrega de m_2 e m_3 pois $cb_{m_4} = \{m_2, m_3\}$. Observe que como m_1 é uma dependência indireta de m_4 (já que m_1 precede m_3), ela não é incluída na barreira causal de m_4. Ainda na Figura 1, o nodo p_3 entra no sistema depois que as mensagens m_2 e m_3 foram difundidas. No entanto, p_3 pode entregar m_4 pois ele ainda não é considerado um assinante correto de t. Apenas após essa primeira entrega, p_3 torna-se um assinante correto e passa a respeitar a ordem causal.

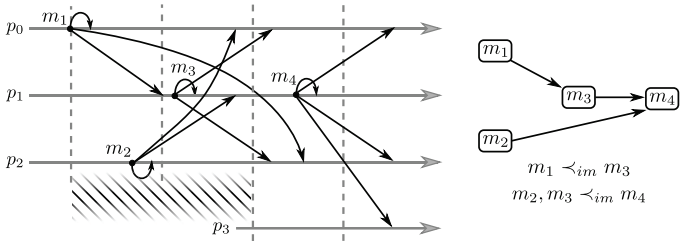


Figura 1. Barreira Causal.

5. Implementação do Sistema VCube-PB

Esta seção apresenta, inicialmente, um resumo breve do *VCube*. Em seguida, descrevemos como adaptamos o princípio do *VCube* para organizar assinantes de um mesmo tópico em diferentes topologias de hipercubo virtual (*VCube-Top*). Por fim, descrevemos os algoritmos utilizados pelo *VCube-PB* para publicação e entrega de mensagem.

5.1. VCube

O *VCube* é uma topologia baseada em hipercubo virtual de dimensão d, criada e mantida com base nas informações de diagnóstico obtidas por meio de um sistema de monitoramento de processos descrito em Duarte Jr. et al. (2014). No *VCube*, cada processo testa outros processos no sistema para verificar se estão corretos ou falhos. Um processo é considerado *correto* se a resposta ao teste for recebida corretamente dentro do intervalo de tempo esperado. Caso contrário, o processo é considerado *falho*. Como o sistema é síncrono, a detecção de falhas é perfeita, ou seja, não há falsas suspeitas.

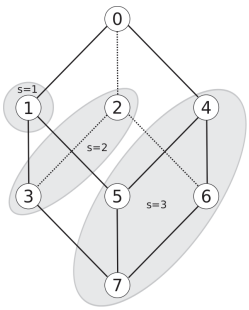
Os processos são organizados em *clusters* progressivamente maiores. Cada *cluster* $s = 1, .., \log_2 n$ possui 2^{s-1} elementos, sendo n o total de processos no sistema. Os testes

são executados em rodadas. Para cada rodada, um processo i testa o primeiro processo sem-falha j na lista de processos de cada *cluster* s e obtém de j as informações que este possui sobre os demais processos do sistema.

Seja $\oplus$ o símbolo que representa a operação binária OU-Exclusivo (XOR). Os membros de cada *cluster* s e a ordem na qual eles são testados por um processo i são obtidos da lista gerada pela seguinte função $c_{i,s}$:

$$c_{i,s} = (i \oplus 2^{s-1}, c_{i\oplus 2^{s-1},1}, ..., c_{i\oplus 2^{s-1},s-1})$$

A Figura 2 exemplifica a organização hierárquica dos processos em um hipercubo de três dimensões com $n = 2^3$ elementos. A tabela da direita contém os elementos de cada *cluster* $c_{i,s}$. Como exemplo, na primeira rodada de testes, o processo p_0 testa: p_1, primeiro processo no *cluster* $c_{0,1} = (1)$, o processo p_2, primeiro processo no *cluster* $c_{0,2} = (2, 3)$ e, por fim, p_4 do *cluster* $c_{0,3} = (4, 5, 6, 7)$. Cada um destes processos envia a p_0 as informações que possuem sobre o estado dos demais processos do sistema. Como todo processo executa estes procedimentos de forma concorrente, ao final da última rodada todo processo será testado ao menos uma vez por um outro processo, garantindo assim que, em $\log_2^2 n$ rodadas (latência de diagnóstico), todos os processos terão localmente a informação atualizada sobre o estado dos demais processos no sistema.



s	$c_{0,s}$	$c_{1,s}$	$c_{2,s}$	$c_{3,s}$	$c_{4,s}$	$c_{5,s}$	$c_{6,s}$	$c_{7,s}$
1	1	0	3	2	5	4	7	6
2	2 3	3 2	0 1	1 0	6 7	7 6	4 5	5 4
3	4 5 6 7	5 4 7 6	6 7 4 5	7 6 5 4	0 1 2 3	1 0 3 2	2 3 0 1	3 2 1 0

Figura 2. Organização Hierárquica do VCube de $d = 3$ dimensões com os *clusters* do processo 0 (esquerda) e a tabela completa da $c_{i,s}$ (direita).

5.2. VCube-Top: Estendendo o VCube para Implementar o Sistema Pub/Sub

No sistema de *Pub/Sub* proposto, os $n = 2^d$ nodos que compõem o sistema não são posicionados em um único hipercubo como no *VCube* original, mas em múltiplos, ou seja, os nodos são logicamente e dinamicamente organizados em uma topologia de hipercubo por tópico e esta não é necessariamente completa.

O nodo i adere ou retira-se do hipercubo referente ao tópico t, como assinante e/ou editor, invocando respectivamente as funções VCUBE_JOIN(t, $tipo$) ou VCUBE_LEAVE (t, $tipo$), oferecidas pelo *VCube-Top*, sendo $tipo$ igual a $subscriber$ ou $publisher$. Note que contrariamente ao *VCube*, um mesmo nodo pode participar de diferentes hipercubos e integrar/retirar-se de um mesmo hipercubo inúmeras vezes. Novas inscrições e cancelamentos de assinaturas de um nodo a um tópico são armazenados localmente e na próxima rodada de monitoramento do *VCube-Top*, as modificações na composição dos assinantes deste tópico são difundidas aos demais assinantes. O conjunto de mudanças é então notificado à camada superior. As seguintes funções definidas em [Rodrigues et al. 2014] foram também estendidas para incluir o tópico:

- $cluster_i(j) = s$: determina a qual *cluster* s do processo i o processo j pertence.
- $FF\_neighbor_i(t, s) = j$: retorna o primeiro processo j correto do *cluster* $c_{i,s}$ que assinou o tópico t. Se todos os processos de $c_{i,s}$ estão falhos ou não são assinantes de t, a função retorna $\perp$.
- $neighborhood_i(t, k) = \{j \mid j = FF\_neighbor_i(t, s), 1 \leq s \leq k\}$: gera, para o tópico t, um conjunto que contém todos os processos sem-falha virtualmente conectados ao processo i de acordo com $FF\_neighbor_i(t, s)$, para $s = 1, .., k$. Por exemplo, no hipercubo sem falhas da Figura 2 se todos os nodos são assinantes de t, $neighborhood_0(t, 3) = \{1, 2, 4\}$, $neighborhood_0(t, 2) = \{1, 2\}$, $neighborhood_4(t, 2) = \{5, 6\}$, etc.

5.3. Algoritmos do VCube-PB

O Algoritmo 5.1 contém as funções oferecidas à aplicação. Um nodo que deseja se registrar apenas como assinante de um tópico t invoca a função SUBSCRIBE(t). Para deixar de receber as mensagens do referido tópico invoca UNSUBSCRIBE(t). Antes de iniciar a publicação no tópico t, o nodo deve se inscrever no mesmo invocando PUBLISH_REGISTER(t), que o adiciona na lista $ptopics_i$. Para publicar uma mensagem m relacionada ao tópico t, basta invocar função PUBLISH(t, m). Essa, por sua vez, após verificar que o nodo está registrado como editor do tópico (linha 14), invoca o procedimento CO_BROADCAST(m) do Algoritmo 5.2. Além dessa função, o Algoritmo 5.2 inclui os procedimentos responsáveis pelo encaminhamento de m a todos os assinantes de t por meio da árvore de difusão do tópico e, a função CO_DELIVER(m) que entrega m à aplicação, assegurando a ordem causal entre as mensagens deste mesmo tópico. O Algoritmo 5.3 contém funções e procedimentos auxiliares utilizados pelo Algoritmo 5.2. Detalhes desse algoritmo serão omitidos por questões de espaço.

Algoritmo 5.1 Interface do VCube-PB no nodo i

```
1:  procedure SUBSCRIBE(topic t)
2:      VCUBE_JOIN(t,"subscriber")
3:      stopics_i ← stopics_i ∪ {t}

4:  procedure UNSUBSCRIBE(topic t)
5:      VCUBE_LEAVE(t,"subscriber")
6:      stopics_i ← stopics_i ∖ {t}

7:  procedure PUBLISHER_REGISTER(topic t)
8:      VCUBE_JOIN(t,"publisher")
9:      ptopics_i ← ptopics_i ∪ {t}

10: procedure PUBLISHER_UNREGISTER(topic t)
11:     VCUBE_LEAVE(t,"publisher")
12:     ptopics_i ← ptopics_i ∖ {t}

13: function PUBLISH(topic t, message m)
14:     if t ∉ ptopics_i then
15:         return NOK
16:     m.source ← i
17:     m.topic ← topic_tag(t)
18:     m.counter ← counter_i[t]
19:     counter_i[t] ← counter_i[t] + 1
20:     CO_BROADCAST(m)
21:     return OK
```

Atualização da composição dos assinantes: todo processo armazena a composição atual do grupo referente a cada tópico em que é assinante ou editor na variável local $members[MAX\_TOPICS]$, que é indexada por tópico. Assim, a cada rodada de testes do *VCube-Top*, este informa (*upon notifying memeberhip* ou *upon notifying crash*, Algoritmo 5.2) as últimas modificações referentes à composição dos *assinantes* dos tópicos: novos registros, cancelamentos ou falhas de nodos.

Difusão de mensagem: cada mensagem m publicada por um nodo i (ver função PUBLISH(t,m), Algoritmo 5.1) é identificada por um *timestamp* único, que contém a identidade do editor ($source$), o tópico ($topic$) associado e um número de sequência ($counter$). A função CO_BROADCAST(m) adiciona a mensagem m em uma fila indexada

Algoritmo 5.2 Difusão causal confiável executada em um processo i

```
1:  upon notifying MEMBERSHIP(join, leave)
2:      for all ⟨j,t⟩ ∈ join do
3:          members_i[t] ← members_i[t] ∪ {j}
4:          stopics_i ← stopics_i ∪ {t}
5:      for all ⟨j,t⟩ ∈ leave do
6:          CHECKLEAVE(j,t)

7:  procedure CO_BROADCAST(message m)
8:      t ← topic(m)
9:      if counter(m) = 1 then
10:         create Task T1(t)
11:     queue_i[t] ← queue_i[t] ∪ {m}

12: Task T1(topic t)
13: loop
14:     m ← queue_i[t].first()          ▷ remoção bloqueante
15:     s ← source(m); t ← topic(m); c ← counter(m)
16:     wait until acks_i ∩ ⟨⊥, *, ⟨s,t,c−1⟩⟩ = ∅
17:     msgs_i ← msgs_i ∪ {m}
18:     if ∄⟨s,*⟩ ∈ first_rec_i[t] then
19:         first_rec_i[t] ← first_rec_i[t] ∪ {⟨s,c⟩}
20:     CO_DELIVER(m)
21:     delvs_i[t] ← delvs_i[t] ∖ {⟨s,c−1⟩} ∪ {⟨s,c⟩}
22:     m.cb ← causal_barrier_i[t]
23:     for all j ∈ neighborhood_s(t, log2 n) do
24:         acks_i ← acks_i ∪ {⟨⊥, j, ⟨s,t,c⟩⟩}
25:         SEND(⟨TREE, m⟩) to p_j
26:     causal_barrier_i[t] ← {⟨s,c⟩}

27: upon RECEIVE ⟨ACK, ⟨s,t,c⟩⟩ from p_j
28:     k ← x : ⟨x, j, ⟨s,t,c⟩⟩ ∈ acks_i
29:     acks_i ← acks_i ∖ {⟨k, j, ⟨s,t,c⟩⟩}
30:     if k ≠ ⊥ then
31:         CHECKACKS(k, ⟨s,t,c⟩)

32: upon RECEIVE ⟨TREE, m⟩ from p_j
33:     s ← source(m); t ← topic(m)
34:     c ← counter(m); cb ← causal_barrier(m)
35:     if ∄⟨s,*⟩ ∈ first_rec_i[t] then
36:         first_rec_i[t] ← first_rec_i[t] ∪ {⟨s,c⟩}
37:     if (⟨s,c,cb⟩ ∉ recs_i[t])
          and (∄ ⟨s,c'⟩ ∈ delvs_i[t] : c' ≥ c) then
38:         recs_i[t] ← recs_i[t] ∪ {⟨s,c,cb⟩}
39:         msgs_i ← msgs_i ∪ {m}
40:         CHECKRECS(t)
41:         if s ∉ members_i[t] then
42:             CO_BROADCAST(m)
43:             return
44:     for all k ∈ neighborhood_i(t, cluster_i(j) − 1) do
45:         if ⟨j,k,⟨s,t,c⟩⟩ ∉ acks_i then
46:             acks_i ← acks_i ∪ {⟨j,k,⟨s,t,c⟩⟩}
47:             SEND(⟨TREE, m⟩) to p_k
48:     CHECKACKS(j, ⟨s,t,c⟩)

49: upon notifying CRASH(j)
50:     for all t' ∈ stopics_i do
51:         CHECKLEAVE(j,t')
```

por tópico. Para cada tópico que um nodo é editor existe também uma tarefa $T1$ associada. Essa tarefa inclui em m o conjunto cb, composto pelas mensagens, identificadas por $\langle source, counter\rangle$, que representam a barreira causal de m. Havendo mensagens na fila para t, $T1(t)$ realiza uma difusão FIFO confiável por meio de uma árvore geradora construída dinamicamente e hierarquicamente, com raiz em i, sobre o hipercubo virtual composto pelos membros interessados no tópico associado à mensagem. Para tanto, as funções descritas na Seção 5.2 são utilizadas. Ao invocar a função $neighborhood_i(t, log_2 n)$, o processo i recebe um conjunto com a identificação dos primeiros processos assinantes de t, não-falhos, de cada um de seus *clusters*, que passam a ser considerados como seus filhos na árvore, e envia então a mensagem m a estes (linha 23). Ao receber m, todo filho j de i envia m aos processos $neighborhood_j(t, cluster_j(i) - 1)$, $cluster_j(i) > 1$, (linha 44) e assim sucessivamente.

Por exemplo, considere a Figura 2 sem nodos falhos e que todos os nodos são assinantes do tópico t_1. Nodo p_0 publica uma mensagem m referente a t_1, se tornando assim a raiz da árvore geradora. A mensagem será enviada aos filhos de p_0 : $FF\_neighbor_0(t_1, 1) = 1$, $FF\_neighbor_0(t_1, 2) = 2$ e $FF\_neighbor_0(t_1, 3) = 4$. Ao receber m, p_1 não a retransmite visto que $cluster_1(0) = 1$. O nodo p_2 recebe m ($cluster_2(0) = 2$) e a retransmite a seu filho p_3, o primeiro nodo do seu *cluster* 1 ($c_{2,1}$). Quando p_3 recebe m, como $cluster_3(2) = 1$, p_3 não retransmite m. No caso de p_4 ($cluster_4(0) = 3$), m é recebida e retransmitida aos seus filhos $p_5 \in c_{4,1}$ e $p_6 \in c_{4,2}$. Finalmente, sendo $cluster_6(0) = 2$, o nodo p_6 envia a mensagem para o processo p_7. Considere agora um segundo exemplo na mesma figura em que apenas os nodos $\{p_0, p_2, p_4, p_5, p_6\}$ são assinantes do tópico t_2 e que p_2 publica a mensagem m' referente

Algoritmo 5.3 Funções Auxiliares

```
1:  function CHECKCB(topic t, causal barrier cb)
2:      for all ⟨s, c⟩ ∈ cb do
3:          if (∃ ⟨s', c'⟩ ∈ delvs_i[t]: s = s' and c' ≥ c) or ( ∃ ⟨s', c'⟩ ∈ first_rec_i[t]: s = s' and c' > c) then
4:              cb ← cb ∖ {⟨s, c⟩}
5:      return (cb = ∅)

6:  procedure CHECKRECS(topic t)
7:      repeat
8:          delMsg ← 0
9:          for all s ← s', c ← c', cb ← cb' : ⟨s', c', cb'⟩ ∈ recs_i[t] do
10:             if CHECKCB(t, cb) then
11:                 recs_i[t] ← recs_i[t] ∖ {⟨s, c, cb⟩}
12:                 delvs_i[t] ← delvs_i[t] ∪ {⟨s, c⟩}
13:                 CO_DELIVER(m), m ∈ msgs_i : source(m) = s, topic(m) = t, counter(m) = c
14:                 causal_barrier_i[t] ← causal_barrier_i[t] ∖ cb ∪ {⟨s, c⟩}
15:                 if ∃ m' ∈ msgs_i: source(m') = s and topic(m') = t and #(m') > c then
16:                     msgs_i ← msgs_i ∖ {m}
17:                 delMsg ← delMsg + 1
18:     until (delMsg = 0)

19: procedure CHECKACKS(process j, ⟨s, t, c⟩)
20:     if acks ∩ ⟨j, ∗, ⟨s, t, c⟩⟩ = ∅ then
21:         if j ∈ members_i[t] then
22:             SEND(⟨ACK, ⟨s, t, c⟩⟩) to p_j

23: procedure CHECKLEAVE(process j, topic t)
24:     members_i[t] ← members_i[t] ∖ {j}
25:     if members_i[t] = ∅ then
26:         stopics_i ← stopics_i ∖ {t}
27:         return
28:     first_rec_i[t] ← ∅
29:     k ← FF_neighbor_i(t, cluster_i(j))
30:     for all x ← x', y ← y', s ← s', c ← c' : ⟨x', y', ⟨s', t, c'⟩⟩ ∈ acks_i do
31:         if {s, x} ⊈ members_i[t] then
32:             acks_i ← acks_i ∖ ⟨x, y, ⟨s, t, c⟩⟩
33:         else if q = j then
34:             if k ≠ ⊥ and ⟨x, k, ⟨s, t, c⟩⟩ ∉ acks_i then
35:                 acks_i ← acks_i ∪ {⟨x, k, ⟨s, t, c⟩⟩}
36:                 m ← m' ∈ msgs_i: source(m') = s, topic(m') = t, counter(m') = c
37:                 SEND(⟨TREE, m⟩) to p_k
38:             acks_i ← acks_i ∖ {⟨x, j, ⟨s, t, c⟩⟩}
39:             CHECKACKS(x, ⟨s, t, c⟩)
40:     if (c' ← max c : ⟨j, c, ∗⟩ ∈ recs_i[t]) ≠ ⊥ then
41:         m ← m' ∈ msgs_i: source(m') = j, topic(m') = t, counter(m') = c'
42:         CO_BROADCAST(m)
```

a t_2. Neste caso, o nodo p_2 envia m' para o primeiro processo sem falha de cada *cluster*, interessado no tópico t_2: $FF\_neighbor_2(t_1, 1) = \perp$ (não há assinantes de t_2 no *cluster* $c_{2,1}$), $FF\_neighbor_2(t_2, 2) = 0$ e $FF\_neighbor_2(t_1, 3) = 6$ (p_6 é o primeiro assinante sem-falha, interessado em t_2 de $c_{2,3}$). Ao receber m', p_0 não a retransmite à p_1 pois este não é assinante de t_2. O nodo p_6 verifica que, no *cluster* $c_{6,2} = (4, 5)$, p_4 também é assinante de t_2, enviando-lhe assim m'. Por fim, p_4 retransmite m' para p_5, contido no *cluster* $c_{4,1}$, e assinante de t_2.

Como a função CO_BROADCAST garante uma entrega confiável, cada processo, após enviar m a seus filhos, aguarda destes uma confirmação (mensagem tipo ACK). Um processo somente envia um ACK a seu pai depois de ter recebido um ACK de todos os seus filhos (função CHECKACKS, Algoritmo 5.3). Para o controle dos $ACKs$ pendentes, todo processo i utiliza a variável conjunto $acks_i$ em que cada elemento guarda informação sobre a mensagem enviada ($\langle s, t, c\rangle$ sendo s a identidade do nodo editor, t o tópico

e, c o número de sequência), o emissor da mensagem j e a qual nodo k a mensagem foi retransmitida. Os dados da mensagem são armazenados na variável conjunto $msgs_i$. Se o *VCube-Top* enviar uma notificação ao processo i informando que ocorreu uma mudança no conjunto de assinantes do tópico devido à falha de k (*upon notifying crash*) ou cancelamento de sua assinatura ao tópico t (*upon notifying membership*), a função CHECKLEAVE (Algoritmo 5.3) é invocada. Neste caso, se i enviou m ao processo k mas ainda não recebeu o ACK de k, i retransmite m para o próximo processo sem-falha do *cluster* de k que é assinante de t. Considere o exemplo anterior na Figura 2 com os assinantes de t_2, mas que p_4 falhe após o envio de m' pelo nodo p_6. Ao detectar a falha de p_4, p_6 enviará m' a p_5, que se torna o próximo processo sem-falha com interesse em t_2 no *cluster* $c_{6,2}$.

Observe que o editor de uma mensagem envia uma nova mensagem referente ao mesmo tópico t somente após receber um ACK de todos os seus filhos (linha 16 do Algoritmo 5.2), assegurando assim uma difusão FIFO. Ao receber esta nova mensagem, um nodo pode excluir do seu conjunto $msgs_i$ a mensagem anterior do mesmo tópico, emitida por este editor. Além disso, se o editor de m falhar, todo processo correto que detectou a falha irá fazer uma nova difusão de m, através da árvore geradora do editor (linha 42 do Algoritmo 5.2 e linha 42 do Algoritmo 5.3), assegurando assim a propriedade do *acordo* (ver Seção 4).

Entrega de mensagem: Ao receber uma mensagem m de j referente ao tópico t, o processo i inclui m no conjunto $recs_i[t]$ e no conjunto $first\_rec_i[t]$, caso m seja a primeira mensagem recebida de j por i para este tópico. O processo i verifica então todas as mensagens recebidas que podem ser entregues à aplicação. Para tanto, i invoca a função CHECKRECS (Algoritmo 5.3) que recursivamente verifica as dependências causais com o auxílio da função CHECKCB. O processo i entrega, por meio da chamada da função CO_DELIVER (m), todas as mensagens cuja causalidade direta pode ser satisfeita, assim como aquelas cuja causalidade não precisa ser satisfeita, pois i se tornou assinante do tópico t após o envio destas mensagens. Esta verificação é possível com o auxílio do valor armazenado em $first\_rec_i[t]$ e pelo fato da difusão de mensagens por um mesmo editor ser FIFO: se i recebe m com barreira causal cb e $\exists\ \langle source, counter\rangle \in cb$ e $\exists\ \langle source, counter'\rangle \in first\_rec_i[t]$, tal que $counter' > counter$, m pode ser entregue imediatamente sem respeitar a ordem definida em cb. Todas as mensagens entregues à aplicação são transferidas de $recs_i[t]$ para $delvs_i[t]$. Como já descrito, ao receber m, i a retransmite aos seus filhos na árvore geradora em questão.

6. Avaliação Experimental

Nesta seção são apresentados e discutidos os resultados experimentais obtidos em diferentes cenários para o *VCube-PB*. As simulações foram implementadas utilizando o *framework* PeerSim [Montresor e Jelasity 2009] para simulações de sistemas distribuídos massivamente dinâmicos.

Os cenários de testes variam em número de assinantes e editores e também na frequência com que os nodos registram ou cancelam assinaturas em tópicos. Considera-se que nodos não falham e um único tópico. Os testes visam à avaliar a escalabilidade da solução proposta, bem como o efeito da causalidade entre as mensagens e o impacto da dinamicidade devido a novas assinaturas e cancelamentos de assinaturas.

Foram realizadas 40 simulações para cada cenário. Os resultados obtidos para latência variam de acordo com o tempo de transmissão de uma mensagem por salto (*hop*), o

tempo total até o recebimento por um assinante e o tempo de entrega da mensagem às camadas superiores desde a sua criação. A cada salto na árvore de difusão, uma mensagem tem seu tempo de transmissão definido seguindo uma distribuição uniforme entre 10.0 e 20.0 unidades de tempo. Além disso, todo nodo possui uma capacidade de processamento de 1 mensagem por unidade de tempo. Caso o nodo receba múltiplas mensagens em um mesmo momento, uma única é processada por vez, respeitando a ordem de chegada.

Um nodo editor sem dinamicidade de assinaturas: inicialmente, para mostrar as propriedades logarítmicas do *VCube-PB*, são avaliados cenários com um único editor por tópico em cinco diferentes cenários de distribuição dos nodos assinantes. Estes são estaticamente distribuídos no início da simulação. No primeiro cenário, todos os nodos são assinantes. No segundo cenário, apenas os vizinhos do nodo editor são assinantes do tópico. No terceiro, os assinantes correspondem ao grupo formado pelo primeiro nodo de cada nível da árvore de difusão. Por fim, são avaliados dois cenários com distribuição aleatória de 25% e 50% dos nodos como assinantes.

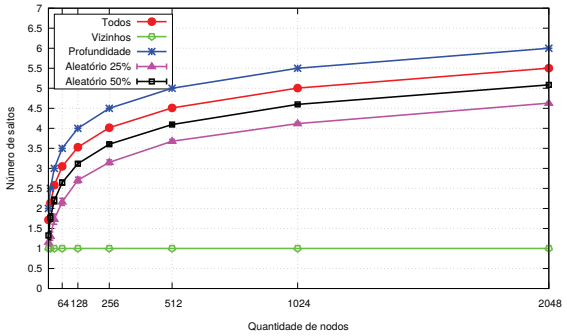


Figura 3. Média de saltos para entrega de uma mensagem a todos os assinantes.

A Figura 3 apresenta a latência média em saltos para que todos os assinantes recebam a mensagem publicada de acordo com os cinco cenários descritos anteriormente. Os dados variam de acordo com o número de nodos no sistemas. Exceto pela curva na qual apenas os vizinhos imediatos do editor estão inscritos, todas as demais possuem um comportamento logarítmico em relação à quantidade de nodos presentes. Isso mostra que a topologia mantém sua característica logarítmica mesmo quando o hipercubo não é completo. Para os cenários mais próximos de uma situação realística, em que os nodos inscritos são escolhidos de forma aleatória, o número médio de saltos é inferior, visto que é possível que a árvore de difusão formada possua menos níveis.

Vários nodos editores com dinamicidade de assinaturas: Contrariamente aos cenários anteriores em que satisfazer a causalidade entre mensagens se resume a entregá-las na ordem FIFO, assegurar a causalidade entre mensagens de diferentes nodos editores faz com que os atrasos na entrega de mensagens sejam mais frequentes e acentuados. Logo, o *overhead* em termos de latência afeta ainda mais o tempo necessário da entrega das mensagens à aplicação.

Nos mesmos cenários acima descritos, o *VCube-PB* foi comparado com uma abordagem com árvore fixa (ex. Scribe [Castro et al. 2006] e Marshmallow [Gao et al. 2011]) na qual o mesmo nodo é sempre a raiz da árvore de difusão, independente do nodo editor. Assim, quando um editor deseja disseminar uma mensagem, ele a envia a esse nodo, que será responsável pela difusão. Isso faz com que todas as mensagens devam necessaria-

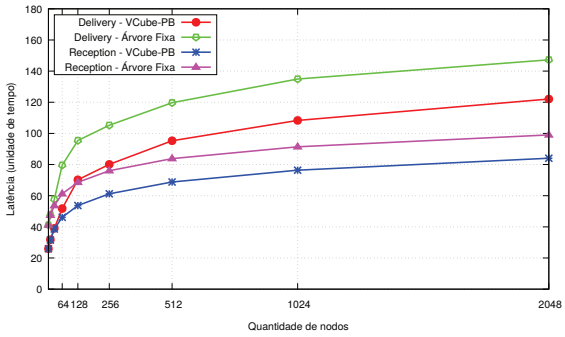
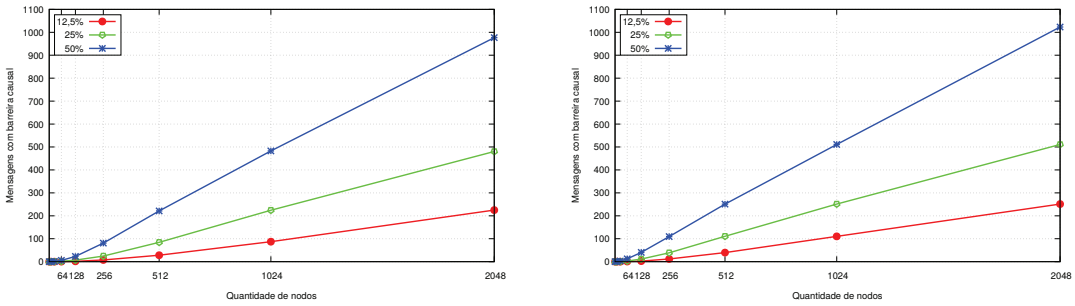


Figura 4. Latência média para recepção e entrega de mensagens.

mente seguir o mesmo caminho e, caso a vazão de mensagens seja alta, a capacidade de processamento do nodo impacta na latência. Por outro lado, para a abordagem proposta com árvores dinâmicas, a posição em que um nodo encontra-se em cada árvore varia de acordo com a sua posição no hipercubo criado pelo *VCube-PB* para o tópico, distribuindo a carga de mensagens, já que cada nodo tem sua árvore de difusão organizada de forma diferente.

A Figura 4 apresenta essa comparação, na qual 25% dos nodos publicam uma única mensagem aleatoriamente em um intervalo de 1000.0 unidades de tempo. As curvas representam as latências médias de recepção e entrega. Também, para aferir a estabilidade da solução proposta em casos de mudança nas assinaturas, a cada 100.0 unidades de tempo ocorre uma variação de até 5% dos nodos inscritos no tópico. Mesmo com o aumento da latência de entrega, podemos observar que o *VCube-PB* apresenta melhores resultados quando comparado com a abordagem de árvore fixa: a latência de entrega do *VCube-PB* tem uma queda entre 37% (8 nodos) e 17% (2048 nodos). Para os resultados exibidos na Figura 4 também foram calculados os intervalos de confiança com nível de confiança em 99%. Como os valores obtidos nunca ultrapassam 4% da média, estes são omitidos da figura.



(a) *VCube-PB* (b) Árvore Fixa

Figura 5. Número de mensagens com dependências causais.

Embora haja uma diminuição da latência quando da utilização de árvores dinâmicas com raiz no nodo editor, o número de mensagens que são afetadas pela causalidade se mantém próximo para as duas abordagens, como retratado nas Figuras 5(a) e 5(b). Em redes com baixo número de nodos, para todos os casos, a probabilidade de que as mensagens tenham seu tempo de entrega afetado pela causalidade é pequena, dado o intervalo

em que são enviadas. Por outro lado, conforme aumenta-se o tamanho da rede, a variação no tempo de cada salto na transmissão das mensagens implica em cada vez mais dependências. Também, ao se saturar a rede com mais nodos publicando em um mesmo intervalo de tempo (50% dos nodos), o número de mensagens afetadas se aproxima do total de mensagens enviadas. O que gera o ganho em termos de latência para a solução proposta é o fato de que o uso de múltiplas árvores distribui a carga do sistema em diferentes partes da rede. Ao se usar uma única árvore por tópico, como o existente no estado da arte, aumenta-se o impacto da capacidade de processamento do nodo na latência de recepção de mensagens. Além disso, o uso de uma única árvore por tópico implica na existência de um ponto de falha no sistema, enquanto que para *VCube-PB*, o editor é raiz de sua própria árvore e, em caso de alterações no conjunto de assinantes, a árvore pode se adaptar mesmo durante a difusão de uma mensagem.

7. Conclusão

Este trabalho apresentou o *VCube-PB*, uma solução distribuída de *publish/subscribe* baseada em tópicos. Mensagens publicadas referentes a cada tópico são transmitidas a todos os assinantes deste tópico por meio de um algoritmo de difusão confiável causal (*reliable causal broadcast*) que utiliza árvores geradoras construídas e reorganizadas dinamicamente sobre a topologia de hipercubo virtual *VCube-Top*, garantindo assim as propriedades logarítmicas do hipercubo mesmo em caso de falhas. Esta também é responsável pela manutenção da visão que um assinante/editor tem dos grupos de assinantes dos tópico nos quais ele também está interessado.

Resultados de avaliação de desempenho sobre o simulador *PeerSim* mostram o comportamento logarítmico do *VCube-PB*, mesmo em situações em que o hipercubo não é completo ou quando há alterações no conjunto de nodos assinantes durante a difusão de mensagens. O uso de múltiplas árvores dinâmicas, embora apresente um número similar de mensagens afetadas pela ordem causal, supera em latência todos os cenários nos quais emprega-se uma única árvore fixa para cada tópico. Como o tempo de processamento das mensagens é um fator limitante no desempenho do sistema, a solução proposta neste trabalho garante uma melhor distribuição da carga de mensagens ao utilizar diferentes árvores de difusão. Além disso, ela elimina o ponto de falha existente ao usar um nodo como raiz fixa para difusão de mensagens em um tópico.

Como trabalhos futuros, além da especificação formal dos algoritmos, serão realizados novos experimentos incluindo cenários com falhas de nodos e uso de *traces* reais.

Agradecimentos

Este trabalho teve apoio do Programa GDE (Doutorado Pleno no Exterior) do CNPq, da Fundação Araucária/SETI (Convênio 144/2015-45112) do projeto 309143/2012-8 do CNPq.

Referências

Baldoni, R., Querzoni, L. e Virgillito, A. (2005). Distributed event routing in Publish/Subscribe communication systems: a survey. Technical report, MIDLAB, Departament of Informatics and Systems, Univerisity of Rome.

Canas, C., Pacheco, E., Kemme, B., Kienzle, J. e Jacobsen, H. (2015). Graps: A graph Publish/Subscribe middleware. Middleware '15, pp. 1–12.

Carzaniga, A., Rosenblum, D. e Wolf, A. (2001). Design and evaluation of a wide-area event notification service. *ACM Trans. Comput. Syst.*, 19(3):332–383.

Castro, M., Druschel, P., Kermarrec, A. M. e Rowstron, A. I. (2006). Scribe: A large-scale and decentralized application-level multicast infrastructure. *IEEE J.Sel. A. Commun.*, 20(8):1489–1499.

Cugola, G., Nitto, E. D. e Fuggetta, A. (2001). The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. *IEEE Trans. Softw. Eng.*, 27(9):827–850.

Duarte, E., Bona, L. e Ruoso, V. (2014). VCube: A provably scalable distributed diagnosis algorithm. In: *5th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, pp. 17–22.

Esposito, C., Cotroneo, D. e Russo, S. (2013). Survey on reliability in Publish/Subscribe services. *Comput. Netw.*, 57(5):1318–1343.

Eugster, P., Felber, P., Guerraoui, R. e Kermarrec, A. (2003). The many faces of Publish/Subscribe. *ACM Comput. Surv.*, 35(2):114–131.

Gao, S., Li, G. e Zhao, P. (2011). Marshmallow: A content-based publish-subscribe system over structured p2p networks. In: *7th CIS*, pp. 290–294.

Google (2016). CLOUD PUB/SUB: A global service for real-time and reliable messaging and streaming data. Disponível em: https://cloud.google.com/pubsub.

Hinze, A. e Buchmann, A., editores (2010). *Principles and Applications of Distributed Event-Based Systems*. IGI Global.

Montresor, A. e Jelasity, M. (2009). Peersim: A scalable p2p simulator. In: *2009 IEEE Ninth International Conference on Peer-to-Peer Computing*, pp. 99–100.

Pietzuch, P. R. e Bacon, J. (2002). Hermes: A distributed event-based middleware architecture. In: *Proc. 22nd Int'l Conf. Distr. Comp. Sys. (ICDCSW'02)*, pp. 611–618.

Raynal, M. (2013). *Distributed Algorithms for Message-Passing Systems*. Springer Publishing Company, Incorporated.

Rodrigues, L., Arantes, L. e Duarte, E. (2014). An autonomic implementation of reliable broadcast based on dynamic spanning trees. In: *10th EDCC*, pp. 1–12.

Rowstron, A. e Druschel, P. (2001). Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. Middleware '01, pp. 329–350.

Strom, R., Banavar, G., Chandra, T., Kaplan, M., Miller, K., Mukherjee, B., Sturman, D. e Ward, M. (1998). Gryphon: An information flow based approach to message brokering. In: *Proc. 9th Int'l Symp. on Soft. Reliability Eng.*

TIBCO (2016). TIBCO rendezvous messaging middleware. Disponível em: http://www.tibco.com/products/automation/enterprise-messaging/rendezvous.

Zhuang, S., Zhao, B., Joseph, A., Katz, R. e Kubiatowicz, J. (2001). Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination. NOSSDAV '01, pp. 11–20.