Uma Proposta de Difusão Confiável Hierárquica em Sistemas Distribuídos Assíncronos

Denis Jeanneau[1], Luiz A. Rodrigues[2], Elias P. Duarte Jr.[3] e Luciana Arantes[1]

[1] Sorbonne Universités, UPMC Université. Paris 06
CNRS, Inria, LIP6 – Place Jussieu, 4 – 75005 – Paris, France

[2]Colegiado de Ciência da Computação – Universidade Estadual do Oeste do Paraná
Caixa Postal 801 – 85819-110 – Cascavel – PR – Brasil

[3]Departamento de Informática – Universidade Federal do Paraná
Caixa Postal 19.081 – 81531-980 – Curitiba – PR – Brasil

denis.jeanneau@lip6.fr, luiz.rodrigues@unioeste.br

Abstract. *This paper presents the work in progress on a hierarchical reliable broadcast solution based on the VCube virtual topology that assumes an asynchronous system. This topology is built and dynamically adapts itself with information obtained from an underlying monitoring system. Broadcast messages are disseminated through a spanning tree that is created and dynamically maintained embedded on a VCube. Processes fail by crashing and a fault is assumed to be eventually detected by all correct processes. In particular we discuss how to deal with false suspicions that arise in the asynchronous environment.*

Resumo. *Este trabalho apresenta a versão preliminar de uma solução hierárquica para a difusão confiável de mensagens com base na topologia virtual mantida pelo VCube. A topologia é construída e adaptada dinamicamente com base nas informações de falhas obtidas de um sistema não confiável de monitoramento. As mensagens são propagadas por uma árvore geradora criada dinamicamente sobre os enlaces mantidos pelo VCube. Os processos podem falhar por crash sem recuperação e uma falha é detectada por todos os processos corretos em um tempo finito. Mensagens diferenciadas são utilizadas para tratar falsas suspeitas geradas pela execução em ambiente assíncrono.*

1. Introdução

Um processo em um sistema distribuído utiliza difusão para enviar uma mensagem a todos os outros processos do sistema [Bonomi et al. 2013]. No entanto, se este processo falha durante o procedimento de difusão, alguns processos podem receber a mensagem enquanto outros não. A difusão confiável garante que, mesmo após a falha do emissor, todos os processos corretos recebem a mensagem difundida por ele [Hadzilacos e Toueg 1993].

Algoritmos de difusão tolerante a falhas são normalmente implementados utilizando enlaces ponto-a-ponto confiáveis e primitivas SEND e RECEIVE. Os processos invocam BROADCAST(m) e DELIVER(m) para difundir e receber uma mensagem m para/de outros processos da aplicação, respectivamente. Para incluir tolerância a falhas, um detector de falhas [Chandra et al. 1996] pode ser utilizado para notificar o algoritmo de *broadcast*, que deve reagir apropriadamente quando uma falha é detectada.

Este trabalho apresenta uma proposta de algoritmo de difusão confiável no qual cada processo é alcançado por meio de uma árvore dinâmica construída sobre uma topologia de hipercubo virtual chamada VCube [Duarte et al. 2014]. O sistema é representado por um grafo completo com enlaces confiáveis. Os processos do sistema são organizados em *clusters* progressivamente maiores formando um hipercubo completo quando não há processos falhos. Em caso de suspeitas, os processos considerados corretos são reconectados entre si para manter as propriedades logarítmicas do hipercubo.

O restante do texto está organizado nas seguintes seções. A Seção 2 discute os trabalhos relacionados. A Seção 3 apresenta as definições básicas, o modelo do sistema e o VCube. O algoritmo de *broadcast* confiável proposto é apresentado na Seção 4. A Seção 5 apresenta a conclusão e os trabalhos futuros.

2. Trabalhos Relacionados

Grande parte dos algoritmos de difusão são baseados em árvores geradoras. Schneider et al. (1984) introduziram um algoritmo de difusão tolerante a falhas baseado em árvore no qual a raiz é o processo que inicia a transmissão, ou seja, o remetente. Cada nodo, incluindo o remetente, envia a mensagem para todos os seus sucessores na árvore. Se um processo p que pertence à árvore falhar, outro processo assume a responsabilidade de retransmitir as mensagens que p deveria ter transmitido se estivesse correto. Os processos podem falhar por *crash* e a falha de um processo é detectada por um módulo de detecção de falhas após um intervalo finito, mas não conhecido. Um processo pode enviar uma próxima mensagem somente após a difusão anterior ter sido concluída. No entanto, os autores não descrevem como a detecção de falhas é implementada, tão pouco fornecem um algoritmo para construir e reorganizar a árvore após a falha.

Em Wu (1996), os autores apresentam um algoritmo de difusão tolerante a falhas para hipercubos baseado em árvores binomiais. O algoritmo pode recursivamente regenerar uma subárvore falha, induzida por um nodo com defeito, através de uma das folhas da árvore. No entanto, ao contrário da abordagem proposta neste trabalho, a solução exige o bloqueio do sistema até que a árvore seja reconstruída.

Liebeherr e Beam (1999) apresentam um procolo, chamado HyperCast, que organiza os membros de um grupo *multicast* em um hipercubo lógico usando o código *Gray* para ordená-los. A árvore é sobreposta no hipercubo para evitar implosão de ACKs. O processo com o identificar mais alto é considerado a raiz da árvore. Entretanto, em função de falhas, múltiplos nodos podem considerar a si próprios como raiz e/ou diferentes nodos podem ter visões diferentes sobre a identidade da raiz.

Em Rodrigues et al. (2014) foi apresentada uma solução para *broadcast* confiável utilizando árvores dinâmicas no VCube. O algoritmo permite a propagação de mensagens utilizando múltiplas árvores construídas dinamicamente a partir de cada emissor e que incluem todos os nodos do sistema. Diferente da solução proposta neste trabalho, o modelo é síncrono e o detector de falhas é perfeito.

3. Definições e Modelo do Sistema

Considera-se um sistema distribuído como um conjunto finito P com $n > 1$ processos $\{p_0, .., p_{n-1}\}$ que se comunicam por troca de mensagens. A rede é representada por um grafo completo. No entanto, processos são organizados em uma topologia de hipercubo

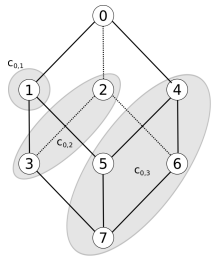
virtual, chamada VCube [Ruoso 2013]. As operações de envio e recebimento são atômicas e os enlaces são confiáveis. O sistema admite falhas de *crash* permanente. Um processo que nunca falha é considerado *correto* ou *sem-falha*. Caso contrário, ele é dito *falho* ou *suspeito*. Considera-se que o VCube implementa um detector de falhas $\diamond S$, isto é, processos falhos são permanentemente suspeitos, mas podem ocorrer falsas suspeitas.

3.1. O VCube

Cada processo que executa o VCube é capaz de testar outros processos no sistema para verificar se estão corretos ou falhos. Os processos são organizados em *clusters* progressivamente maiores. Cada *cluster* $s = 1, .., \log_2 n$ possui 2^s elementos, sendo n o total de processos no sistema. Para cada rodada um processo i testa o primeiro processo sem-falha j na lista de processos de cada *cluster* s e obtém dele as informações que ele possui sobre os demais processos do sistema.

Os membros de cada *cluster* s e a ordem na qual eles são testados por um processo i são obtidos da lista gerada pela função $c_{i,s} = (i \oplus 2^{s-1}, c_{i\oplus 2^{s-1},1}, ..., c_{i\oplus 2^{s-1},s-1})$ ($\oplus$ é a operação binária de OU exclusivo – XOR).

A Figura 1 exemplifica a organização hierárquica dos processos em um hipercubo de três dimensões com $n = 2^3$ elementos. A tabela da direita apresenta os elementos de cada *cluster* $c_{i,s}$. Como exemplo, na primeira rodada o processo p_0 testa o primeiro processo no *cluster* $c_{0,1} = (1)$ e obtém informações sobre o estado dos demais processos armazenada em p_1. Em seguida, p_0 testa o processo p_2, que é primeiro processo no *cluster* $c_{0,2} = (2, 3)$. Por fim, p_0 executa testes no processo p_4 do *cluster* $c_{0,3} = (4, 5, 6, 7)$. Como cada processo executa estes procedimentos de forma concorrente, ao final da última rodada todo processo será testado ao menos uma vez por um outro processo. Isto garante uma latência de diagnóstico máxima de $\log_2^2 n$ rodadas.



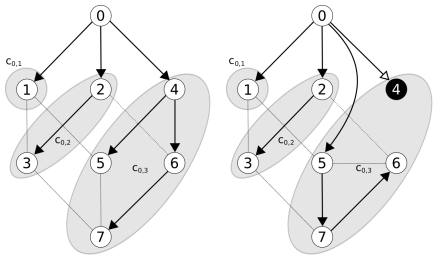
s	$c_{0,s}$	$c_{1,s}$	$c_{2,s}$	$c_{3,s}$	$c_{4,s}$	$c_{5,s}$	$c_{6,s}$	$c_{7,s}$
1	1	0	3	2	5	4	7	6
2	2 3	3 2	0 1	1 0	6 7	7 6	4 5	5 4
3	4 5 6 7	5 4 7 6	6 7 4 5	7 6 5 4	0 1 2 3	1 0 3 2	2 3 0 1	3 2 1 0

Figura 1. Organização Hierárquica do VCube de $d = 3$ dimensões.

4. O Algoritmo de Difusão Confiável Proposto

Seja i um processo que executa o algoritmo de difusão confiável e $d = \log_2 n$ a dimensão do d-VCube com 2^d processos. O Algoritmo 1 apresenta o pseudo-código da solução de difusão confiável proposta.

A função $cluster_i(j) = s$ calcula o identificador s do *cluster* do processo i que contém o processo j, $1 \leq s \leq d$. Por exemplo, considerando o 3-VCube da Figura 1, $cluster_0(1) = 1$, $cluster_0(2) = cluster_0(3) = 2$ e $cluster_0(4) = cluster_0(5) = cluster_0(6) = cluster_0(7) = 3$. Três tipos de mensagens são utilizados: $\langle TREE, m\rangle$ para identificar a mensagem de aplicação m que está sendo propagada na árvore; $\langle DELV, m\rangle$ para as mensagens enviadas aos processos considerados falhos,



(a) Sem falhas (b) p_4 é suspeito

Figura 2. Difusão confiável no processo 0 (p_0)

evitando que falsas suspeitas impliquem em não recebimento por processos corretos; e $\langle ACK, m\rangle$ para confirmar o recebimento de m pelo destinatário que recebe TREE. Cada mensagem m contém ainda dois parâmetros: (1) o identificador da origem, isto é, o processo que iniciou a difusão, obtido com a função $source(m)$; e (2) o *timestamp*, um contador sequencial local que identifica de forma única cada mensagem gerada em um processo, obtido pela função $ts(m)$. Um processo obtém as informações sobre o estado dos demais processos pelo algoritmo VCube.

As variáveis locais mantidas pelos processos são:

- $correct_i$: conjunto dos processos considerados corretos pelo processo i;
- $last_i[n]$: a última mensagem recebida de cada processo fonte;
- ack_set_i: o conjunto de ACKs pendentes no processo i. Para cada mensagem $\langle TREE, m\rangle$ (re)-transmitida por i de um processo j para um processo k, um elemento $\langle j, k, m\rangle$ é adicionado a este conjunto. O símbolo $\perp$ representa um elemento nulo. O asterisco é usado como curinga. Por exemplo, um elemento $\langle j, *, m\rangle$, por exemplo, representa todos os ACKs pendentes para uma mensagem m recebida pelo processo j e retransmitida para qualquer outro processo.
- $pending_i$: lista de mensagens m recebidas pelo processo i de um processo fonte $source(m)$ que ainda não podem ser entregues à aplicação por estarem fora de ordem, isto é, $ts(m) > ts(last_i(source(m)) + 1$.

Um processo i inicia a difusão invocando o método BROADCAST(m). A linha 7 garante que um novo *broadcast* só é iniciado após o término do anterior, isto é, quando não há *acks* pendentes para a mensagem $last_i[i]$. Nas linhas 9 - 10 a nova mensagem m é entregue localmente (propriedade de entrega confiável) e em seguida enviada a todos os vizinhos no VCube através da função BROADCAST_TREE. Esta função utiliza BROADCAST_CLUSTER para enviar mensagens TREE para cada primeiro processo sem-falha em cada cluster $s = 1..\log_2 n$, e mensagens DELV para os processos considerados falhos (suspeitos). Para cada mensagem TREE enviada, um *ack* é incluído na lista de *acks* pendentes. A Figura 2 ilustra exemplos para um VCube de 3 dimensões. A Figura 2(a) mostra uma execução sem falhas considerando o processo 0 (p_0) como fonte. Após fazer a entrega local, p_0 envia uma cópia da mensagem para p_1, p_2 e p_4, que são os vizinhos dele em cada *cluster*. Embora p_5 também pertença ao mesmo *cluster* de p_4, p_0 não envia uma cópia da mensagem para p_5, pois não há aresta no VCube conectando os dois processos.

Quando um processo i recebe uma mensagem $\langle TREE, m\rangle$ de um processo j (linha 39) ele invoca HANDLE_MESSAGE. Nela é verificado se a mensagem é nova comparando os *timestamps* da última mensagem armazenada em $last_i[j]$ e da mensagem recebida m (linha 33), garantindo a propriedade de integridade. Se m é um nova mensagem,

$last_i[j]$ é atualizado e a mensagem é entregue à aplicação. Em seguida, o processo i verifica se o processo origem da mensagem está falho. Se a origem ainda é considerada correta, m é retransmitida para os vizinhos em cada *cluster* interno ao *cluster* de i que também fazem parte do grupo com os destinatários da mensagem. Isso é feito durante a execução da função BROADCAST_CLUSTER com parâmetro $h = cluster_i(j) - 1$. A Figura 2(a) ilustra a propagação feita por p_4 para p_5. Se i é uma folha da árvore (*clusters* $s = 1$) ou se não existe vizinho correto pertencente aos destinatários, nenhum *ack* pendente é adicionado ao conjunto ack_set_i e CHECKACKS envia uma mensagem `ACK` para j. Por outro lado, se um processo i recebe uma mensagem nova $\langle TREE, m\rangle$, mas verifica que $source(m)$ foi detectado como falho, o processo de *broadcast* é reiniciado considerando a árvore com raiz em i.

Quando uma mensagem $\langle ACK, m\rangle$ é recebida, o conjunto ack_set_i é atualizado e, se não existem mais *acks* pendentes para a mensagem m, CHECKACKS envia um `ACK` para o processo k do qual i recebeu a mensagem `TREE` anteriormente. No entanto, se $k = i$, a mensagem `ACK` alcançou o processo fonte ou o processo que retransmitiu a mensagem após a falha do processo fonte. Nesse caso, a mensagem de `ACK` não precisa mais ser propagada.

A detecção de um processo falho j é tratada após a notificação do evento

Algoritmo 1 Difusão confiável no processo i

```
1: last_i[n] ← {⊥, ..., ⊥}
2: ack_set_i ← ∅
3: correct_i ← {0, ..., n − 1}
4: pending_i ← ∅

5: procedure BROADCAST(message m)
6:     if source(m) = i then
7:         wait until ack_set_i ∩ {⟨⊥, ∗, last_i[i]⟩} = ∅
8:         last_i[i] ← m
9:         DELIVER(m)
10:    BROADCAST_TREE(⊥, m, log_2 n)

11: procedure BROADCAST_TREE(process j, message m, integer h)
12:    for all s ∈ [1, h] do
13:        BROADCAST_CLUSTER(j, m, s)

14: procedure BROADCAST_CLUSTER(process j, message m, integer s)
15:    sent ← false
16:    for all k ∈ c_{i,s} do
17:        if sent = false then
18:            if ⟨j, k, m⟩ ∈ ack_set_i and k ∈ correct_i then
19:                sent ← true
20:            else if k ∈ correct_i then
21:                SEND(⟨TREE, m⟩) to p_k
22:                ack_set_i ← ack_set_i ∪ {⟨j, k, m⟩}
23:                sent ← true
24:            else if ⟨j, k, m⟩ ∉ ack_set_i then
25:                SEND(⟨DELV, m⟩) to p_k

26: procedure CHECK_ACKS(process j, message m)
27:    if j ≠ ⊥ and ack_set_i ∩ {⟨j, ∗, m⟩} = ∅ then
28:        if j ∈ correct_i then
29:            SEND(⟨ACK, m⟩) to p_j

30: procedure HANDLE_MESSAGE(message m)
31:    pending_i ← pending_i ∪ {m}
32:    while ∃l ∈ pending_i : (source(l) = source(m) ∧
       ts(l) = ts(last_i[source(m)]) + 1)
33:        or (last_i[source(m)] = ⊥ ∧ ts(l) = 0) do
34:        last_i[source(m)] ← l
35:        pending_i ← pending_i \ {l}
36:        DELIVER(l)
37:    if source(m) ∉ correct_i then
38:        BROADCAST(m)

39: upon receive ⟨TREE, m⟩ from p_j
40:    HANDLE_MESSAGE(m)
41:    BROADCAST_TREE(m, cluster_i(j) − 1)
42:    CHECK_ACKS(j, m)

43: upon receive ⟨DELV, m⟩ from p_j
44:    HANDLE_MESSAGE(m)

45: upon receive ⟨ACK, m⟩ from p_j
46:    for all k = x : ⟨x, j, m⟩ ∈ ack_set_i do
47:        ack_set_i ← ack_set_i \ {⟨k, j, m⟩}
48:        CHECK_ACKS(k, m)

49: upon notifying crash(process j)
50:    correct_i ← correct_i ∖ {j}
51:    for all p = x, m = y : ⟨x, j, y⟩ ∈ ack_set_i ∩
       {⟨∗, j, ∗⟩} do
52:        BROADCAST_CLUSTER(p, m, cluster_i(j))
53:        ack_set_i ← ack_set_i \ {⟨p, j, m⟩}
54:        CHECK_ACKS(p, m)
55:    if last_i[j] ≠ ⊥ then
56:        BROADCAST(last_i[j])

57: upon notifying up(process j)
58:    correct_i ← correct_i ∪ {j}
```

CRASH(j). Três ações são realizadas: (1) atualização da lista de processos corretos; (2) remoção dos *acks* pendentes que contém o processo j como destino ou aqueles em que a mensagem m foi originada em j; (3) reenvio das mensagens anteriormente transmitidas ao processo j para o novo vizinho correto k no mesmo *cluster* de j, se existir um. Esta retransmissão desencadeia uma propagação na nova estrutura da árvore. No exemplo da Figura 2(b), após a notificação de falha de p_4, p_0 retransmite a mensagem para o processo p_5, visto que $c_{0,3} = (4, 5, 6, 7)$, p_5 é o próximo processo sem falha no *cluster* $s = 3$. A propagação continua pelos demais processos corretos do *cluster*, isto é, p_6 e p_7. Se p_4 é considerado falho antes do início da difusão, uma mensagem DELV é enviada a ele para garantir o recebimento por p_4 em caso de falsa suspeita.

5. Conclusão

Este trabalho apresentou uma proposta de solução distribuída para a difusão confiável (*broadcast*) em sistemas distribuídos sujeitos a falhas de *crash* em ambientes assíncronos. Árvores com raiz em cada processo são construídas e mantidas dinamicamente sobre uma topologia de hipercubo virtual denominada VCube. Em caso de falhas, os processos são reorganizados de forma a manter as propriedades logarítmicas do hipercubo. Falsas suspeitas são contornadas pelo envio de mensagens adicionais aos processos considerados falhos, porém mantendo-se as principais propriedades do hipercubo.

Como trabalhos futuros, o algoritmo será especificado formalmente, incluindo provas de correção, e testes de desempenho serão realizados por simulação comparando-o com outras soluções.

Referências

Bonomi, S., Del Pozzo, A. e Baldoni, R. (2013). Intrusion-tolerant reliable broadcast. Technical report, Sapienza Università di Roma,.

Chandra, T. D., Hadzilacos, V. e Toueg, S. (1996). The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722.

Duarte, Jr., E. P., Bona, L. C. E. e Ruoso, V. K. (2014). VCube: A provably scalable distributed diagnosis algorithm. In: *5th Work. on Latest Advances in Scalable Algorithms for Large-Scale Systems*, ScalA'14, pp. 17–22, Piscataway, USA. IEEE Press.

Hadzilacos, V. e Toueg, S. (1993). Fault-tolerant broadcasts and related problems. In: Distributed systems, pp. 97–145. ACM Press, New York, NY, USA, 2 ed.

Liebeherr, J. e Beam, T. (1999). HyperCast: A protocol for maintaining multicast group members in a logical hypercube topology. In: Rizzo, L. e Fdida, S., editores, *Networked Group Communication*, v. 1736 de *LNCS*, pp. 72–89. Springer Berlin Heidelberg.

Rodrigues, L. A., Duarte Jr., E. P. e Arantes, L. (2014). Árvores geradoras mínimas distribuídas e autonômicas. In: *XXXII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*, SBRC'14.

Ruoso, V. K. (2013). Uma estratégia de testes logarítmica para o algoritmo Hi-ADSD. Dissertação de Mestrado, Universidade Federal do Paraná.

Schneider, F. B., Gries, D. e Schlichting, R. D. (1984). Fault-tolerant broadcasts. *Sci. Comput. Program.*, 4(1):1–15.

Wu, J. (1996). Optimal broadcasting in hypercubes with link faults using limited global information. *J. Syst. Archit.*, 42(5):367–380.