Agregação de Mensagens em uma Solução Hierárquica de Difusão de Melhor-Esforço

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

[1] Colegiado de Ciência da Computação – Unioeste-Cascavel, Brasil

[2]Departamento de Informática – UFPR, Brasil

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

luiz.rodrigues@unioeste.br

Abstract. *Best-effort broadcast guarantees that a message is delivered by all processes of a distributed systems, as long as the source does not fail. One of the most important metrics to evaluate broadcast algorithms is the number of sent messages. In this work we present a best-effort broadcast solution that employs message aggregation in order to reduced the number of messages transmitted without increasing the latency to complete the broadcast. Dynamic trees are built and maintained from the source over the VCube, a virtual hypercube-based topology. The messages are aggregated according to the destination across the subtrees, respecting maximum limits of packet size and delay at each node. Simulation results are presented and show the efficiency of the solution in several scenarios with and without crash failures.*

Resumo. *A difusão de melhor esforço garante a entrega de uma mensagem por todos os processos de um sistema distribuído, desde que a origem não falhe. Uma das métricas importantes para avaliar a eficiência da difusão é o número de mensagens envidadas. Neste trabalho é apresentada uma solução para difusão de melhor esforço com agregação de mensagens que visa reduzir o número de mensagens utilizadas sem prejudicar a latência da difusão. Árvores dinâmicas são construídas e mantidas a partir origem sobre o VCube, uma topologia virtual baseada em hipercubo. As mensagens são agregadas de acordo com seus destinos nas subárvores, respeitando limites máximos de tamanho dos pacotes e atraso em cada nodo. Resultados de simulação demonstram a eficiência da solução em diversos cenários com e sem falhas por parada.*

1. Introdução

Um processo em um sistema distribuído utiliza difusão (*broadcast*) para enviar uma mensagem a todos os outros processos do sistema. A difusão de melhor esforço (*best-effort broadcast*) garante que todos os processos entregam a mensagem caso a origem não falhe [Hadzilacos e Toueg 1993]. Este trabalho apresenta um algoritmo de difusão de melhor-esforço com agregação de mensagens no qual cada processo do sistema é alcançado por meio de árvores dinâmicas mantidas sobre a topologia do VCube [Duarte Jr. et al. 2014]. Cada árvore é construída em tempo de propagação a partir do processo fonte, isto é, aquele que inicia a difusão. Por utilizar a mesma topologia, árvores com raiz em diferentes processos possuem diversos ramos sobrepostos, o que permite a agregação das mensagens

transmitidas para um mesmo destino (intermediário ou final) da árvore. O sistema é representado logicamente por um grafo completo com enlaces confiáveis.

Uma estratégia para difusão de melhor-esforço utilizando o VCube foi apresentada anteriormente em [Rodrigues et al. 2014]. O presente trabalho apresenta um novo algoritmo que utiliza agregação de mensagens. O algoritmo proposto foi implementado em simulador e comparado com a versão sem agregação, em cenários com e sem falhas por parada. Os resultados confirmam a eficiência da solução proposta considerando: (1) a latência para entregar a mensagem a todos os processos corretos; (2) o número total de mensagens, incluindo retransmissões em caso de falhas; e (3) tamanho das mensagens.

O restante do texto está organizado nas seguintes seções. A Seção 2 discute os trabalhos correlatos. A Seção 3 apresenta o modelo do sistema e o VCube. O algoritmo de difusão de melhor esforço proposto é apresentado na Seção 4. Os resultados de simulação são apresentados na Seção 5. A Seção 6 apresenta a conclusão e os trabalhos futuros.

2. Trabalhos Relacionados

Grande parte das soluções que envolvem agregação de dados está ligada ao contexto de redes de sensores, voltadas principalmente à eficiência energética e latência [Rajagopalan e Varshney 2006, Weng et al. 2008, Son et al. 2006]. Neste caso, o objetivo é agrupar dados redundantes em uma única mensagem para entregá-las ao coletor (*sink*). O trabalho de Villas et al. (2010), por exemplo, propõe uma técnica para construção de árvores dinâmicas de agregação de dados em redes de sensores sem fio que minimizam o número de mensagens redundantes entre os emissores e os coletores.

Em [Khanna et al. 2002] foi proposto um *framework* de agregação de mensagens de controle para ambientes *multicast* e protocolos hierárquicos. A agregação é feita com base na quantidade máxima de mensagens e no limite temporal de espera de cada mensagem que compõe o pacote. Diferente da solução proposta neste trabalho, mensagens de aplicação não são agregadas e o impacto da agregação em árvores com estrutura conhecida é apontado como um trabalho em aberto. No trabalho de [Hidalgo et al. 2010], a agregação de mensagens é utilizada no contexto de P2P para reduzir o tráfego de rede e o número de saltos para a entrega das mensagens.

Em Rodrigues et al. (2014) foi apresentada uma solução para *broadcast* 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. Uma restrição daquele trabalho era permitir que um novo *broadcast* só fosse iniciado após o término do anterior.

3. Definições e Modelo do Sistema

Considera-se um sistema distribuído composto por 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 com enlaces ponto-a-ponto bidirecionais. No entanto, processos são organizados em uma topologia de hipercubo virtual, chamada VCube [Duarte Jr. et al. 2014] (mais detalhes na Seção 3.1). As operações de envio e recebimento são atômicas. Enlaces são confiáveis, garantindo que as mensagens nunca são perdidas, corrompidas ou duplicadas pelos enlaces.

O sistema admite falhas de parada de processos (*crash*), que são permanentes. Um processo que nunca falha é considerado *correto* ou *sem-falha*. Caso contrário ele é considerado *falho*. O sistema é síncrono, isto é, existem limites conhecidos para a velocidade de processamento e atraso de transmissão de mensagens. Neste modelo, o VCube garante uma detecção de falhas perfeita, não admitindo falsas suspeitas [Freiling et al. 2011].

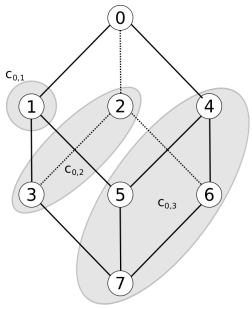
3.1. O VCube

O VCube é uma topologia baseada em hipercubo virtual 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]. Cada processo que executa o VCube é capaz de testar 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. Os processos são organizados em *clusters* progressivamente maiores. Cada *cluster* $s = 1, .., \log_2 n$ possui 2^{s-1} elementos. 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 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 $c_{i,s}$, definida a seguir. O símbolo $\oplus$ representa a operação binária de OU exclusivo (XOR):

$$c_{i,s} = i \oplus 2^{s-1} \parallel c_{i\oplus 2^{s-1},k} \mid k = 1, .., s-1 \tag{1}$$

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 que em $\log_2^2 n$ rodadas, todos os processos terão localmente a informação atualizada sobre o estado dos demais processos no sistema (latência de diagnóstico).



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 com os *clusters* do processo 0 (esq.) e a tabela completa da $c_{i,s}$ (dir.)

Seja i um processo que executa o algoritmo proposto e $d = \log_2 n$ a dimensão do d-VCube com 2^d processos. A função $cluster_i(j) = s$ retorna o índice 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 $cluster_0(4) = cluster_0(5) = cluster_0(6) = cluster_0(7) = 3$.

4. Difusão de Melhor-Esforço com Agregação de Mensagens

A difusão de melhor-esforço garante a entrega das mensagens de um processo emissor (fonte) correto a todos os demais processos corretos do sistema. Três propriedades caracterizam este modelo: entrega confiável (*validity*), não-duplicação e não-criação de mensagens [Guerraoui e Rodrigues 2006]. A entrega confiável garante que, se um processo i envia uma mensagem m para um processo j e nenhum deles falha, j recebe m em um tempo finito. A não-duplicação garante que nenhuma mensagem é entregue mais de uma vez e a não-criação garante que nenhuma mensagem é entregue a menos que tenha sido previamente enviada.

O Algoritmo 1 apresenta uma solução para difusão de melhor-esforço hierárquico com agregação de mensagens que utiliza o mecanismo de árvore gerado a partir do VCube, que também é utilizado para obter as informações sobre o estado dos demais processos do sistema. O algoritmo executa corretamente mesmo que $n-1$ falhas ocorram.

Cada mensagem m da aplicação contém dois atributos: (1) o identificador da origem, isto é, o processo que iniciou a difusão, obtido pelo atributo $m.src$; e (2) o *timestamp*, um contador sequencial local iniciado em zero que identifica de forma única cada mensagem gerada em um processo, obtido pelo atributo $m.ts$.

Dois tipos de mensagens são utilizados:

- $\langle TREE, m \rangle$: identifica a mensagem de aplicação m que está sendo propagada;
- $\langle ACK, (src, ts) \rangle$: confirma o recebimento de m pelo destinatário, sendo $src = m.src$ e $ts = m.ts$.

Dois parâmetros são necessários para a operação do algoritmo: $maxDelay$, que indica o atraso máximo de uma mensagem enquanto aguarda a agregação; e $maxPayload$, que define o tamanho máximo do pacote com as mensagens agregadas.

As variáveis locais mantidas pelos processos são:

- $correct_i$: conjunto dos processos considerados corretos pelo processo i;
- $rcv\_base_i[n]$: *timestamp* da última mensagem entregue em ordem de cada processo fonte;
- $rcv\_win_i[n]$: listas das mensagens entregues fora de ordem de cada processo fonte;
- ack_set_i: o conjunto com todos os ACKs pendentes no processo i. Para cada mensagem $\langle TREE, m \rangle$ recebida pelo processo i de um processo j e retransmitida para o processo k, um elemento $\langle j, k, (m.src, m.ts) \rangle$ é adicionado a este conjunto;
- msg_set_i: conjunto de mensagens com *ack* pendente no processo i, utilizado para retransmissões em caso de falhas;
- $delay\_msg_i[n]$: listas das mensagens agregadas aguardando para serem enviadas para o mesmo processo destino;
- $timer_i[n]$: contador de tempo que controla a espera máxima para envio das mensagens agregadas. A função STOP $timer_i[j]$ desliga o temporizador do processo j e a função START $timer_i[j]$ inicia o temporizador do processo j com o valor do parâmetro $maxDelay$.

O símbolo $\perp$ representa um elemento nulo. O asterisco é usado como curinga para selecionar ACKs no conjunto ack_set. 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.

Considerando a organização dos processos no sistema conforme apresentado na seção 3, define-se a função $FF\_neighbor_i(s) = j$ (***FF=Fault-Free***) que identifica o primeiro processo j no *cluster* s do processo i, isto é, j é o primeiro processo da lista gerada pela $c_{i,s}$. Por exemplo, considerando a tabela da Figura 1, $FF\_neighbor_0(1) = 1$, $FF\_neighbor_0(2) = 2$ e $FF\_neighbor_0(3) = 4$. Em caso de falha do processo 2, $FF\_neighbor_0(2) = 3$.

Em seguida, define-se a função $subtree_i(j)$ como sendo o conjunto de processos vizinhos de i em relação ao processo de origem j (*source*) da mensagem sendo propagada. Tal função é definida como:

$$subtree_i(j) = \{\forall k = FF\_neighbor_i(s)\} \begin{cases} s = 1..\log_2 n, \text{se } i = j \\ s = 1..(cluster_i(j) - 1), \text{se } i \neq j \end{cases} \quad (2)$$

Algoritmo 1 Difusão de melhor-esforço hierárquica com agregação de mensagens em p_i

Entrada: $maxDelay$ ▷ Intervalo máximo de atraso da mensagem em cada nodo
Entrada: $maxPayload$ ▷ Tamanho máximo do conjunto de mensagens agregadas

1: **upon** Initialization
2: $ack\_set_i \leftarrow \emptyset$ ▷ Conjunto de acks pendentes
3: $msg\_set_i \leftarrow \emptyset$ ▷ Conjunto de mensagens de com ack pendente
4: $correct_i = \{0, .., n-1\}$ ▷ Conjunto dos processos considerados corretos
5: $\forall j \in 0..n-1 : rcv\_base_i[j] \leftarrow \perp$ ▷ *ts* da última mensagem de j entregue em ordem
6: $\forall j \in 0..n-1 : rcv\_win_i[j] \leftarrow \emptyset$ ▷ Mensagens de j entregues fora de ordem
7: $\forall j \in FF\_neighbor_i(s) \mid s = 1..\log_2 n : delay\_msg_i[j] = \emptyset, timer_i[j]$ é desligado

8: **procedure** BROADCAST(mensagem m)
9: HANDLE_MESSAGE($\langle TREE, m \rangle$, i)

10: **procedure** HANDLE_MESSAGE(mensagem $\langle TREE, m \rangle$, processo j)
11: $msg\_set_i \leftarrow msg\_set_i \cup \{m\}$
12: **if** $rcv\_base_i[m.src] = \perp$ **or**
13: $(m.ts > rcv\_base_i[m.src]$ **and** $m \notin rcv\_win_i[m.src])$ **then**
14: DELIVER(m)
15: UPDATE_WINDOW(m)
16: FORWARD($\langle TREE, m \rangle$, j)
17: CHECK_ACKS($(m.src, m.ts)$, j)

18: **procedure** UPDATE_WINDOW(mensagem m)
19: **if** $(rcv\_base_i[m.src] = \perp$ **then**
20: **if** $m.ts = 0$ **then**
21: $rcv\_base_i[m.src] \leftarrow m.ts$
22: **else**
23: $rcv\_win_i[m.src] \leftarrow rcv\_win_i[m.src] \cup \{m\}$

24: **else if** $m.ts = rcv\_base_i[m.src] + 1$ **then**
25: $rcv\_base_i[m.src] \leftarrow m.ts$
26: **else**
27: $rcv\_win_i[m.src] \leftarrow rcv\_win_i[m.src] \cup \{m\}$
28: **while** $\exists\, m' \in msg\_win[m.src] \mid m'.ts = rcv\_base_i[m.src] + 1$ **do**
29: $rcv\_base_i[m.src] \leftarrow m'.ts$
30: $rcv\_win_i[m.src] \leftarrow rcv\_win_i[m.src] \setminus \{m'\}$

31: **procedure** FORWARD(mensagem $\langle TREE, m\rangle$, processo j)
32: **for all** $k \in subtree_i(m.src)$ **do** ▷ Encaminha m para todos os vizinhos sem-falha
33: CHECK_AGGREGATION($\langle TREE, m\rangle$, k)
34: $ack\_set_i \leftarrow ack\_set_i \cup \{\langle j, k, (m.src, m.ts)\rangle\}$

35: ▷ $\langle T, c\rangle$: a mensagem $T = TREE \mid ACK$ que será agregada
36: **procedure** CHECK_AGGREGATION(mensagem $\langle T, c\rangle$, processo k)
37: **if** $length(delay\_msg_i[k] \cup \langle T, c\rangle) > maxPayload$ **then**
38: SEND($delay\_msg_i[k]$) **to** p_k
39: $delay\_msg_i[k] \leftarrow \{\langle T, c\rangle\}$
40: stop $timer_i[k]$
41: **else if** $length(delay\_msg_i[k] \cup \langle T, c\rangle) = maxPayload$ **then**
42: SEND($delay\_msg_i[k] \cup \{\langle T, c\rangle\}$) **to** p_k
43: $delay\_msg_i[k] \leftarrow \emptyset$
44: stop $timer_i[k]$
45: **else**
46: $delay\_msg_i[k] \leftarrow delay\_msg_i[k] \cup \{\langle T, c\rangle\}$
47: **if** $timer_i[k]$ is off **and** $delay\_msg_i[k] \neq \emptyset$ **then**
48: start $timer_i[k]$

49: **procedure** CHECK_ACKS(ack (src, ts), processo k)
50: **if** $ack\_set_i \cap \{\langle k, *, (src, ts)\rangle\} = \emptyset$ **then**
51: $msg\_set_i \leftarrow msg\_set_i \setminus \{m\} : m.src = src, m.ts = ts$
52: **if** $k \neq i$ **and** $\{src, k\} \subseteq correct_i$ **then**
53: CHECK_AGGREGATION($\langle ACK, (src, ts)\rangle$, k)

54: **upon** RECEIVE (set<$\langle T, c\rangle$> $msgs$) **from** p_j
55: **for all** $\langle T, c\rangle \in msgs$ **do**
56: **if** $T = TREE$ **then**
57: RECEIVE($\langle TREE, m = c\rangle$, j)
58: **else if** $T = ACK$ **then**
59: RECEIVE($\langle ACK, (src, ts) = c\rangle$, j)

60: **procedure** RECEIVE(mensagem $\langle TREE, m\rangle$, processo j)
61: **if** $\{m.src, j\} \nsubseteq correct_i$ **then**
62: **return**
63: HANDLE_MESSAGE($\langle TREE, m\rangle$, j)

64: **procedure** RECEIVE(mensagem $\langle ACK, (src, ts)\rangle$, processo j)

```
65:     k ← x : ⟨x, j, (src, ts)⟩ ∈ ack_set_i
66:     ack_set_i ← ack_set_i ∖ {⟨k, j, (src, ts)⟩}
67:     CHECK_ACKS((src, ts), k)

68: upon timer_i[k] expiration
69:     SEND(delay_msg_i[k]) to p_k
70:     delay_msg_i[k] ← ∅

71: upon notifying CRASH(processo j)                    ▷ j é detectado como falho
72:     correct_i ← correct_i ∖ {j}
73:     delay_msg_i[j] ← ∅; stop timer_i[j]
74:     k ← FF_neighbor_i(cluster_i(j))
75:     for all p = x, q = y, (src, ts) = z : ⟨x, y, z⟩ ∈ ack_set_i do
76:         if {src, p} ⊈ correct_i then
77:                                  ▷ Remove acks pendentes para ⟨j, ∗, ∗⟩ e ⟨∗, ∗, (src, ts)⟩ : src = j
78:             ack_set_i ← ack_set_i ∖ {⟨p, q, (src, ts)⟩}
79:         else if q = j then          ▷ Envia m para o novo vizinho sem-falha k (se existir um)
80:             if k ≠ ⊥ and ⟨p, k, (src, ts)⟩ ∉ ack_set_i then
81:                 ack_set_i ← ack_set_i ∪ {⟨p, k, (src, ts)⟩}
82:                 m ← m′ ∈ msg_set_i : m′.src = src, m′.ts = ts
83:                 CHECK_AGGREGATION(⟨TREE, m⟩, k)
84:             ack_set_i ← ack_set_i ∖ {⟨p, j, (src, ts)⟩}
85:         CHECK_ACKS((src, ts), p)
```

Um processo i que deseja propagar uma mensagem m invoca o método BROADCAST, que repassa m para o método HANDLE_MESSAGE. Neste ponto, $m.src$ já carrega o identificador do processo fonte e $m.ts$ contém o *timestamp* da mensagem, inicializado em zero e incrementado em um para cada novo *broadcast*. O tamanho total da mensagem (ou conjunto de mensagens) é calculado pela função $length(m)$ com base nas informações recebidas da aplicação e dos campos adicionais inseridos pela camada de difusão.

Toda mensagem $\langle TREE, m\rangle$ encaminhada é armazenada temporariamente em msg_set_i (linha 11) enquanto existir *ack* pendente para m, possibilitando a sua retransmissão em caso de falhas (linha 82). Em seguida, na linha 13, é verificado se a mensagem m nunca foi entregue (*delivered*) ao processo i, garantindo a propriedade de não-duplicação. Esta verificação é feita por meio de duas variáveis locais: rcv_base_i e rcv_win_i. Primeiro, é verificado se m é a primeira mensagem recebida de $m.src$, isto é, $rcv\_base_i[m.src] = \perp$; em seguida é verificado se o *timestamp* de m é maior que o da última mensagem entregue em ordem ou se m não está na lista $rcv\_win_i[m.src]$ de mensagens entregues fora de ordem. Se m é a primeira mensagem recebida, verifica-se se o *timestamp* é zero, o que indica que m é a primeira mensagem enviada por $m.src$ e $rcv\_base_i[m.src]$ é atualizado com $m.ts$. Se $m.ts > 0$, a primeira mensagem recebida não é a primeira enviada pela origem e, portanto, ela é entregue e incluída na lista de mensagens entregues fora de ordem, evitando assim qualquer entrega duplicada no futuro. Se m não é a primeira mensagem recebida de $m.src$, primeiro verifica-se se $m.ts$ é imediatamente superior ao *timestamp* da última mensagem entregue em ordem, isto é, $m.ts = rcv\_base_i[m.src] + 1$. Neste caso, $rcv\_base_i[m.src] = m.ts$. Caso contrário, m será entregue fora de ordem e armazenada em $rcv\_win_i[m.src]$, evitando

assim futuras entregas duplicadas. Os casos em que m preenche uma lacuna entre a última mensagem recebida em ordem e as armazenadas em rcv_win_i são resolvidos nas linhas 28-30, com a atualização de $rcv\_base_i[m.src]$ com o *timestamp* da última mensagem recebida em ordem armazenada em $rcv\_win_i[m.src]$ e a remoção da mesma da lista de mensagens entregues fora de ordem. Independente da entrega local, na linha 32 a mensagem recebida de j é encaminhada a todos os vizinhos k considerados corretos por meio do método FORWARD. Para cada mensagem enviada, um *ack* $\langle j, k, (m.src, m.ts)\rangle$ é incluído na lista de *acks* pendentes. Se não existe vizinho correto ou se i é uma folha na árvore ($cluster_i(j) = 1$), nenhum *ack* pendente é adicionado ao conjunto ack_set_i e CHECK_ACKS envia um ACK para j.

Quando um processo recebe uma mensagem $TREE$ de um processo j (linha 60), ele primeiramente verifica se tanto o processo fonte da mensagem quanto o processo j são considerados corretos. Se um deles está falho, o recebimento é abortado, pois se j está falho, o processo que transmitiu m para j fará uma nova transmissão quando detectar a falha e i irá receber a mensagem através da nova árvore que será reconstruída. Além disso, se o fonte está falho, não é mais necessário continuar a retransmissão. Se o fonte e j são considerados corretos, o processo i verifica se a mensagem é nova utilizando o mesmo processo descrito anteriormente, definido na função HANDLE_MESSAGE.

Quando uma mensagem $\langle ACK, (src, ts)\rangle$ é recebida, o conjunto ack_set_i é atualizado e, se não existem mais ACKs pendentes para a mensagem $m : m.src = src, m.ts = ts$, CHECK_ACKS remove a mensagem m referente ao *ack* recebido do conjunto msg_set_i. Se $k \neq i$, um $\langle ACK, (src, ts)\rangle$ é enviado para o processo k do qual i recebeu a mensagem $TREE$ anteriormente. Se $k = i$, o ACK alcançou o processo fonte e não precisa mais ser propagado.

A detecção de um processo falho j é tratada no evento de CRASH. Três ações importantes 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 j para o novo vizinho k no mesmo cluster de j, se existir um. Esta retransmissão desencadeia uma nova propagação na estrutura da árvore reconstruída.

O método CHECK_AGGREGATION é responsável por verificar a possibilidade de agregação das mensagens enviadas pelo processo i, tanto TREEs quanto ACKs. Com base no tamanho de cada mensagem m, é verificado se o conjunto de mensagens aguardando envio $delay\_msg_i[k]$ para um destino k não ultrapassa o tamanho máximo do pacote definido por $maxPayload$. Se a inclusão da nova mensagem ultrapassa o tamanho máximo (linha 37), o conjunto atual é enviado e a nova mensagem é enfileirada para uma futura transmissão e o relógio $timer_i[k]$ é reiniciado assumindo o valor de $maxDelay$. Se a nova mensagem completa o pacote de tamanho máximo (linha 41), ela é enviada em conjunto com as mensagens que aguardavam em $delay\_msg_i[k]$ e o relógio é desativado, uma vez que não existem mais mensagens pendentes. Em último caso, se a nova mensagem não ultrapassa e nem completa o tamanho máximo, ela é apenas incluída no conjunto das mensagens que aguardam o envio. O tempo máximo que uma mensagem aguarda para ser enviada é controlado pelo temporizador $timer_i[k]$. Sempre que o temporizador de um destino k expira (linha 68), as mensagens pendentes em $delay\_msg_i[k]$ são imediatamente enviadas, todas em um mesmo pacote.

O tamanho de uma mensagem é definido em função do conteúdo que ela carrega. As mensagens $TREE$ possuem, além do conteúdo da mensagem de aplicação (APP), um cabeçalho H, composto por um código de identificação id e pelos campos obrigatórios descritos anteriormente: identificador do processo de origem (src) e o *timestamp* (ts). As mensagens ACK possuem também um cabeçalho com o id os campos necessários para identificar a mensagem que está sendo confirmada, também representados pelo par (src, ts). Desta forma, o tamanho das mensagens $TREE$ e ACK difere basicamente pelo conteúdo da mensagem de aplicação, podendo ser calculados por:

$$\begin{aligned} H &= \langle id, (src, ts) \rangle \\ length(ACK) &= length(H) \\ length(TREE) &= length(H) + length(APP) \end{aligned} \tag{3}$$

O tamanho do pacote com as mensagens agregadas é dado pela soma dos tamanhos das mensagens $TREE$ e ACK que ele carrega.

5. Avaliação Experimental

Nesta seção são apresentados os resultados dos experimentos de simulação realizados com o algoritmo de difusão com agregação proposto comparando-os com a versão sem agregação. Os algoritmos foram implementados utilizando o Neko [Urbán et al. 2002]. Os testes estão divididos em duas partes. Primeiro são apresentados os resultados para cenários sem processos falhos e, em seguida, para os cenários com falhas.

5.1. Parâmetros de Simulação

Quando um processo precisa enviar uma mensagem para mais de um destinatário ele deve utilizar primitivas SEND sequencialmente. Assim, para cada mensagem enviada, t_s unidades de tempo são utilizadas para processar o envio da mensagem e t_r unidades para processar o recebimento dela. Há ainda o atraso de transmissão t_t, utilizado pelo enlace para propagar a mensagem no canal de comunicação. Estes intervalos são computados para cada cópia da mensagem enviada pelo processo fonte e em cada salto na topologia virtual do hipercubo.

Para avaliar o desempenho de soluções de difusão, três métricas foram utilizadas: (1) a latência para entregar a mensagem de *broadcast* a todos os processos corretos; (2) total de mensagens enviadas pelo algoritmo, que incluem as mensagens `TREE` e `ACK`; e (3) tamanho das mensagens em bytes. Todos os processos realizam a difusão de uma única mensagem no início da simulação.

O algoritmo proposto foi avaliado em diferentes cenários variando-se o número de processos e, nos cenários com falhas, a quantidade de processos falhos. Os parâmetros de comunicação foram definidos em $t_s = t_r = 0.1$ e $t_t = 0.8$. O intervalo de testes do detector é de 30.0 unidades de tempo. Além disso, um processo é considerado falho se não responder ao teste após $4 * (t_s + t_r + t_t)$ unidades de tempo, isto é, 4.0.

Foram geradas cinco configurações diferentes variando-se os seguintes parâmetros: (1) tamanho máximo do pacote com mensagens agregadas; (2) atraso máximo para envio das mensagens agregadas; (3) tamanho das mensagens $TREE$; e (4) tamanho das

mensagens ACK. Os valores dos parâmetros em cada cenário estão listados na Tabela 1, na qual cada coluna representa um dos cenários.

No cenário (NO-AGGR), nenhuma mensagem é agregada. Para evitar ajustes no algoritmo, o tamanho máximo do pacote e das mensagens $TREE$ e ACK foi definido em 1, e o atraso máximo é zero. Nos demais cenários, variou-se o tamanho das mensagens e o atraso de envio. Nos cenários SMALL, o tamanho de $TREE$ é 50, ligeiramente maior que o de ACK que é 34, representando aplicações que propagam poucas informações. Para os cenários BIG, o tamanho de ACK continua sendo 34 e as mensagens $TREE$ possuem tamanho 500. O atraso máximo foi definido em 2 ou 10 unidades de tempo, indicada pelo valor correspondente. No cenário SMALL2, por exemplo, o tamanho máximo do pacote é 1480, as mensagens $TREE$ possuem tamanho 50 e o atraso máximo é 2. Já no cenário BIG10, o tamanho máximo do pacote é 1480, as mensagens $TREE$ possuem tamanho 500 e o atraso máximo é 10. Pra efeitos de comparação, o cenário NO-AGGR assume tamanhos de mensagens equivalentes ao cenário comparado.

Tabela 1. Parâmetros de configuração dos cenários de teste.

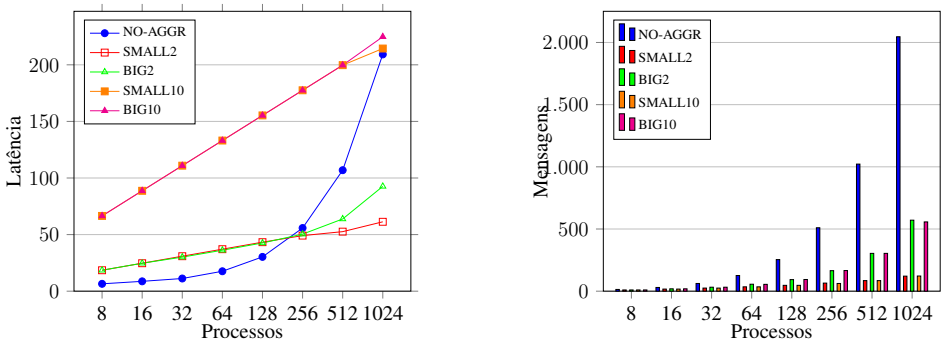
	NO-AGGR	SMALL2	BIG2	SMALL10	BIG10
Tamanho máximo dos pacotes	1	1480	1480	1480	1480
Tamanho da mensagem TREE	1	50	500	50	500
Tamanho da mensagem ACK	1	34	34	34	34
Atraso máximo	0	2	2	10	10

5.2. Cenários sem Falhas

A Figura 2 apresenta os resultados obtidos para sistemas de diferentes tamanhos e sem processos falhos. A latência representa o tempo total para concluir a difusão em todos os processos nos cinco cenários propostos, isto é, o tempo total de simulação considerando que todos os processos realizam um *broadcast*. O número de mensagens representa a média de mensagens enviadas em relação ao número de processos.

O cenário sem agregação apresenta uma menor latência em relação aos testes com agregação em sistemas com menos de 256 processos, embora o número de mensagens seja extremamente maior. A agregação de mensagens de aplicação com tamanho reduzido, representada pelo cenário SMALL2, mostra-se mais vantajosa em relação à latência e número de mensagens que as demais. No entanto, SMALL10 é equivalente em latência a BIG10, embora tenha enviado um número significativamente menor de mensagens. Percebe-se que o atraso incluído no envio das mensagens tem impacto semelhante na latência nos cenários com agregação, independente do número maior ou menor de mensagens agregadas em cada pacote. No entanto, verifica-se que tal impacto tende a diminuir com o aumento da carga do sistema para mensagens menores (SMALL), como pode ser visto nos cenários com 1024 processos.

A Figura 3 compara graficamente as mensagens agregadas enviadas pelo processo p_0 em um sistema com 1024 processos sem falhas nos cenários SMALL2 e BIG2. Em função do tamanho menor das mensagens de aplicação de SMALL2, o número de mensagens agregadas é maior, principalmente no início da simulação. Para BIG2, a agregação tem maior efeito no final da simulação, com o envio dos ACKs, que possuem menor tamanho. Com isso, SMALL2 consegue completar a difusão enviando 121 mensagens



(a) Latência (b) Total de Mensagens

Figura 2. Latência e média de mensagens para difusão de uma mensagem em cada processo em uma execução sem falhas.

(Tabela 3) em 61,3 intervalos de tempo (Tabela 2), ao passo que BIG2 envia 571 mensagens e finaliza em 92,6 intervalo de tempo. Isto representa uma diferença de 372% no número de mensagens e 51,1% no tempo total da difusão.

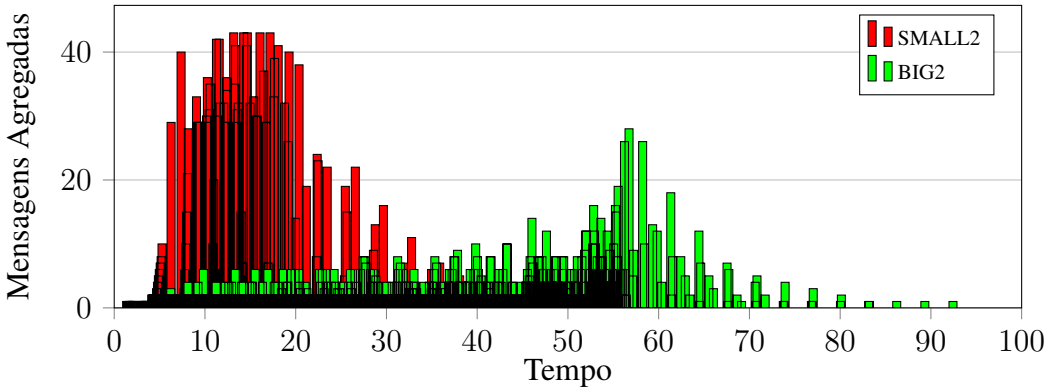


Figura 3. Comparativo do número de mensagens agregadas no processo p_0 nos cenários SMALL2 e BIG2 com 1024 processos em uma execução sem falhas.

Comparando novamente os cenários sem falhas com 1024 processos para SMALL2 e BIG2 em termos do tamanho das mensagens enviadas pelo processo p_0, observa-se na Figura 4 que SMALL2 consegue aproveitar melhor a capacidade máxima do pacote (1480) quando o volume de mensagens é maior. No entanto, BIG2 ainda apresenta um bom aproveitamento em relação à agregação dos ACKs.

Em termos de total de bytes transmitidos, a título de exemplificação, pode-se ainda comparar os cenários SMALL2 e BIG2 com o cenário NO-AGGR. Nos sistemas sem falhas com 1.024 processos são geradas 2.046 mensagens por processo (1023 TREEs e 1023 ACKs). Comparando-se SMALL2 com NO-AGGR utilizando os parâmetros de tamanho de mensagens da Tabela 1, cada processo gera (1.023*50)+(1.023*34)=85.932 bytes na camada de difusão, independente do cenário. No entanto, enquanto NO-AGGR envia as 2.046 mensagens, SMALL2 envia apenas 121. Considerando uma rede com cabeçalho padrão de 20 bytes, isto geraria uma sobrecarga de cabeçalho de 2.046*20=40.920 bytes e 121*20=2.420 bytes, respectivamente. Trata-se de um acréscimo de 1.591% no número de mensagens e de 43,6% no total de bytes transmitidos no enlace. De forma semelhante,

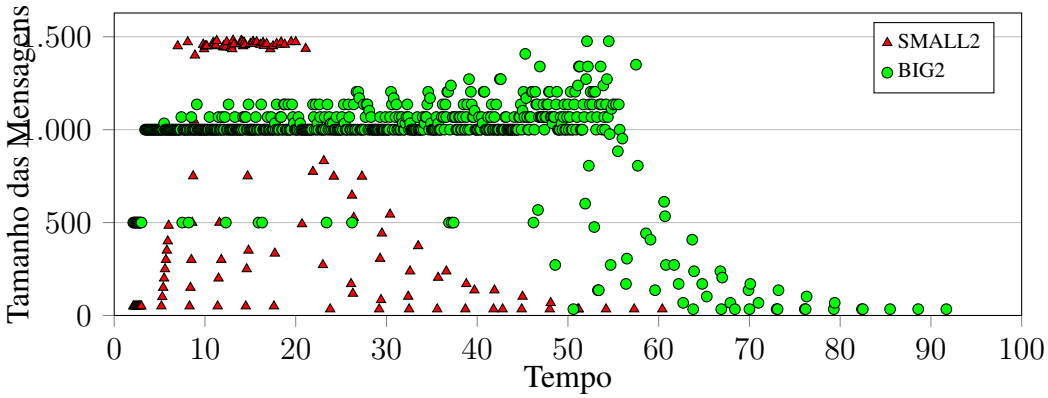


Figura 4. Comparativo do tamanho de mensagens agregadas no processo p_0 nos cenários SMALL2 e BIG2 com 1024 processos em uma execução sem falhas.

BIG2 e NO-AGGR com os mesmos parâmetros geram (1.023*500)*(1.023*34)=546.282 bytes. A sobrecarga de cabeçalho de rede para NO-AGGR é a mesma da comparação anterior e para BIG2 é de 571*20=11.420. Isto representa uma diferença de 258,3% no total de mensagens enviadas por cada processo e de 5,3% no total de bytes transmitidos. Note que a diferença é reduzida em função do melhor aproveitamento do pacote em NO-AGGR com mensagens maiores, o que confirma a vantagem da agregação para aplicações que trocam mensagens pequenas e com grande frequência, como nos cenários SMALL.

5.3. Cenários com Falhas

Para avaliar o impacto de falhas na solução proposta, foram executados experimentos utilizando o mecanismo de colapso (*crash*) disponível no Neko. Dois tipos de testes foram executados. No primeiro, um único processo falha no início da simulação. O segundo teste utiliza falhas aleatórias para demonstrar o funcionamento do algoritmo proposto.

Falha de um único processo. Inicialmente o processo p_1 foi configurado para falhar no tempo $t = 0.0$, isto é, no início da simulação. A falha é detectada pelos processos vizinhos de p_1 no VCube (p_0, p_3 e p_5) e propagada de acordo com as rodadas de teste do VCube. Cada processo, ao detectar a falha de p_1, executa os passos indicados pelo evento CRASH do algoritmo. Novas mensagens $TREE$ podem ser transmitidas de acordo com a reconfiguração do hipercubo.

A Tabela 3 compara a média de mensagens (TREE e ACK) transmitidas por cada processo correto em uma execução sem falhas (FF) e com a falha do processo p_1 (FY). No cenário sem agregação, cada *broadcast* é realizado com uma mensagem a menos, referente ao ACK não devolvido pelo processo falho. Note que no cenário executado, no início da simulação os processos ainda não foram informados pelo VCube sobre a falha de p_1 e, portanto, as mensagens $TREE$ são retransmitidas para ele normalmente. Esta latência de detecção do VCube justifica o aumento do tempo de execução nos cenários com falha registrados na Tabela 2. Embora tenha impacto em todos os cenários, a latência do VCube é mais evidente nos cenários sem agregação (NO-AGGR) ou com baixo atraso de agregação (SMALL2 e BIG2). Nos cenários SMALL10 e BGI10 o atraso de agregação oculta a latência de detecção.

Múltiplas falhas. Por fim, falhas múltiplas foram injetadas em cada cenário, sendo o

Tabela 2. Latência nos cenários sem-falha (FF) e com um processo falho (FY).

Processos	NO-AGGR		SMALL2		BIG2		SMALL10		BIG10	
	FF	FY	FF	FY	FF	FY	FF	FY	FF	FY
8	6,5	12,8	18,5	24,8	18,5	24,8	66,5	66,6	66,5	66,6
16	8,7	15,4	24,7	31,1	24,7	31,1	88,7	88,8	88,7	88,8
32	11,2	18,4	30,9	37,4	30,1	36,5	110,9	111,0	110,9	111,0
64	17,6	23,3	37,1	43,7	36,2	42,7	133,1	133,2	133,1	133,2
128	30,3	35,0	43,3	49,7	42,6	48,8	155,3	155,4	155,3	155,6
256	55,8	66,2	49,1	55,3	50,4	56,9	177,5	177,8	177,5	177,6
512	106,9	125,8	52,6	62,1	63,8	73,0	199,7	199,9	199,7	200,4
1.024	209,2	243,9	61,3	66,4	92,6	104,0	214,4	222,3	224,8	231,8

Tabela 3. Média de mensagens por processo correto nos cenários sem-falha (FF) e com um processo falho (FY).

Processos	NO-AGGR		SMALL2		BIG2		SMALL10		BIG10	
	FF	FY	FF	FY	FF	FY	FF	FY	FF	FY
8	14	13	10	10	10	10	10	8	10	8
16	30	29	17	16	19	18	17	16	19	17
32	62	61	25	25	32	32	25	24	32	31
64	126	125	35	35	56	54	35	34	55	53
128	254	253	47	45	93	92	47	46	94	92
256	510	508	65	61	165	163	62	62	166	162
512	1022	1020	85	86	304	302	85	84	304	301
1.024	2046	2044	121	126	571	569	122	121	557	555

número de processos falhos igual a $(log_2 n) - 1$. Os processos falhos e o tempo em que as falhas acontecem foram gerados aleatoriamente. Cada cenário foi executado com 10 configurações de falhas distintas. A Tabela 4 mostra os resultados obtidos. Diferente dos experimentos anteriores, nos quais foram apresentados as médias de mensagens enviadas por cada processo, neste caso foram computadas todas as mensagens trocadas por todos os processos, visto que o instante em que a falha acontece em cada processo impacta no número de mensagens enviadas por ele e pelos demais processos corretos virtualmente ligados a ele pelo VCube. A latência continua sendo o tempo total de simulação para concluir o *broadcast* dos processos corretos.

Tabela 4. Latência com múltiplas falhas com intervalo de Confiança (IC) de 95%.

	NO-AGGR		SMALL2		BIG2		SMALL10		BIG10	
p	Média	IC	Média	IC	Média	IC	Média	IC	Média	IC
8	22,2	11,5	32,3	10,6	32,3	15,5	72,5	6,2	72,5	6,2
16	35,9	9,8	50,8	10,1	50,8	19,2	109,5	10,8	110,1	10,5
32	62,0	6,0	80,7	5,9	80,7	23,2	154,3	5,9	154,2	6,3
64	63,0	3,5	84,8	3,6	84,8	24,2	176,2	3,8	175,7	4,1
128	70,1	2,4	89,8	2,7	89,7	24,2	196,1	3,0	197,2	3,2
256	85,2	0,4	90,6	0,2	91,5	22,2	212,0	0,3	213,4	0,5
512	129,4	0,1	96,3	0,1	103,5	19,1	233,3	0,0	233,4	0,0
1024	247,0	0,3	102,1	0,0	129,4	14,7	255,3	0,2	267,6	0,3

6. Conclusão

Este trabalho apresentou uma solução distribuída para a difusão de melhor-esforço com agregação de mensagens em sistemas distribuídos síncronos sujeitos a falhas de parada. Múltiplas árvores com raiz em cada processo são construídas e mantidas dinamicamente sobre um VCube, uma topologia escalável baseada em hipercubo virtual, que se adapta automaticamente em caso de falhas. Resultados de simulação comparando a solução proposta com uma abordagem sem agregação mostram a eficiência do mesma em cenários

com e sem falhas, especialmente em relação ao número e ao tamanho das mensagens enviadas. Como trabalhos futuros, pretende-se adaptar o algoritmo para o modelo assíncrono e executar os testes em uma rede real.

Agradecimentos

Estre trabalho teve apoio da Fundação Araucária/SETI (convênio 341/10) e do CNPq, proj. 309143/2012-8.

Referências

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.

Freiling, F. C., Guerraoui, R. e Kuznetsov, P. (2011). The failure detector abstraction. *ACM Comput. Surv.*, 43:9:1–9:40.

Guerraoui, R. e Rodrigues, L., editores (2006). *Introduction to Reliable Distributed Programming*. Springer-Verlag, Berlin, Germany.

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.

Hidalgo, N., Arantes, L., Sens, P. e Bonnaire, X. (2010). An aggregation-based routing protocol for structured peer to peer overlay networks. In: *2nd Int'l Conf. on Advances in P2P Systems (AP2PS)*, pp. 76–81, Florence, Italy. ThinkMind.

Khanna, S., Naor, J. S. e Raz, D. (2002). Control message aggregation in group communication protocolsIn: Widmayer, P., Eidenbenz, S., Triguero, F., Morales, R., Conejo, R. e Hennessy, M., editores, *Automata, Languages and Programming*, pp. 135–146, Berlin, Heidelberg. Springer Berlin Heidelberg.

Rajagopalan, R. e Varshney, P. K. (2006). Data-aggregation techniques in sensor networks: A survey. *IEEE Communications Surveys Tutorials*, 8(4):48–63.

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, pp. 1–14.

Son, J., Pak, J. e Han, K. (2006). In-network processing for wireless sensor networks with multiple sinks and sources. In: *Proc. 3rd Int'l Conf. Mobile Technology, Applications and Systems*, Mobility'06, New York, NY, USA. ACM.

Urbán, P., Défago, X. e Schiper, A. (2002). Neko: A single environment to simulate and prototype distributed algorithms. *Journal of Inf. Science and Eng.*, 18(6):981–997.

Villas, L. A., Guidoni, D. L., Araújo, R. B., Boukerche, A. e Loureiro, A. A. (2010). A scalable and dynamic data aggregation aware routing protocol for wireless sensor networks. In: *Proc. 13th ACM Int'l Conf. on Modeling, Analysis, and Simulation of Wireless and Mobile Systems*, pp. 110–117, New York, NY, USA. ACM.

Weng, C., Li, M. e Lu, X. (2008). Data aggregation with multiple spanning trees in wireless sensor networks. In: *Int'l Conf. on Embedded Software and Systems*, pp. 355–362.