Bundling Messages to Reduce the Cost of Tree-Based Broadcast Algorithms

Luiz A. Rodrigues, Elias P Duarte Júnior, João Paulo de Araujo, Luciana Arantes, Pierre Sens

Bundling Messages to Reduce the Cost of Tree-Based Broadcast Algorithms

Luiz A. Rodrigues*, Elias P. Duarte Jr. † João Paulo de Araujo‡, Luciana Arantes‡ and Pierre Sens‡
*Western Paraná State University, Department of Computer Science, Cascavel, Brazil
E-mail: luiz.rodrigues@unioeste.br
†Federal University of Paraná, Department of Informatics, Curitiba, Brazil
E-mail: elias@inf.ufpr.br
‡ Sorbonne Université, CNRS, INRIA, LIP6, Paris, France
E-mail: {joao.araujo,luciana.arantes,pierre.sens}@lip6.fr

Abstract—**Hierarchical broadcast strategies based on trees are scalable since they distribute the workload among processes and disseminate messages in parallel. In this work we propose an efficient best-effort broadcast algorithm that employs multiple spanning trees to propagate messages that are bundled on tree intersections. The algorithm is autonomic in the sense that it employs dynamic trees rooted at the source process and which rebuild themselves after processes crash. Experimental results obtained with simulation are presented showing the performance to the algorithm in terms of the latency and the number and sizes of messages employed.**

Index Terms—**Distributed systems, Fault-Tolerant Broadcasts, Spanning trees**

1. Introduction

A process of a distributed system uses broadcast to propagate a message to all other processes in the system. Best-effort broadcast ensures that all correct processes deliver the message if the source does not fail. If the source process can fail, another type of broadcast must be employed: reliable broadcast. Furthermore it is possible to define the order in which messages are delivered, such as those guaranteed by FIFO, causal and atomic broadcast [1].

Several broadcast algorithms are based on trees [2], [3], [4], [5], [6]. If a single tree is employed, each message is first sent to the root of the tree, that will start the broadcast. The main problem of this approach is that the root can become a bottleneck. In addition, these solutions need the extra step in which an initial message is sent to the root to start the broadcast. As faults are unavoidable in distributed systems, the algorithms should be able to build and maintain the tree even as faults occur.

One way to mitigate these problems is to use multiple adaptive trees. In [7], [8], [9] we proposed broadcast algorithms based on the VCube [10]. VCube is hypercube-like virtual topology that reconfigures itself and maintains multiple logarithmic properties even after processes crash. The broadcast algorithm propagates messages along spanning trees which are embedded on the VCube. We call the solution *autonomic* in the sense that the broadcast and the trees tolerate and automatically adapt to process crashes.

Whenever multiple trees are built on a given network, if those trees overlap this can be harnessed by bundling together messages that come to the intersections and are supposed to be forwarded to the same routes. Not only the number of messages is reduced but also the overhead of running the algorithm on the network.

In this paper, we discuss and evaluate the impact of bundling messages to reduce the communication cost of tree-based broadcast algorithms. We present a new best-effort broadcast algorithm based on message bundling. The algorithm was implemented, and simulation results are presented which evaluate the benefits of message bundling through a comparison with an alternative without bundling. Both fault-free and faulty (crash) scenarios were considered, with multiple other features, such as the size of messages and the number of processes. Results confirm the efficiency of the proposed solution considering: (i) the latency to deliver the message to all the correct processes; (ii) the total number of messages, including retransmissions in case of failures; and (iii) size of messages.

The rest of this paper is organized as follows. Section 2 gives an overview of related work. Section 3 describes the system model and the VCube, the virtual topology used. The spanning tree algorithm and the motivation for bundling messages are presented in Section 4. The broadcast algorithm is presented in Section 5. Performance results are discussed in Section 6. Section 7 concludes the paper.

2. Related Work

Many distributed systems and application on different research areas such as Peer-to-Peer (P2P) Overlay Networks [11], [12], [13], Internet of Things (IoT) [14], [15], Wireless Sensor Networks (WSN) [16], [17], [18], Parallel Discrete Event Simulation (PDES) [19],among others, employ message aggregation/combination in order to reduce the communication cost and thus improve the performance of their solutions.

In [12], a structured P2P routing protocol combines multiple lookup messages into a single one with the goal of reducing the average number of hops messages traverse. In [11], the identifier space of the P2P system is divided into slices, coordinated by a leader node. The latter collects all membership change notifications sent from the nodes of its slice during a period of time and, aiming at reducing bandwidth usage, aggregates the notifications into a single message before sending them to the other slice leaders. Similarly, in [13], a message bundling technique improves network throughput by reducing the number of packet transmissions and mitigates the load of nodes on the overlay.

The main motivation of data aggregation in WSN and IoT is to save energy by reducing the number of message transmissions. However, this may have an impact on other performance metrics such as latency, load processing, or fault tolerance [16], [17], [18], [20]. In this type of work, the goal is usually to group redundant data into a single message which is delivered to the sink. For example, the authors in [20] present a technique for building dynamic data aggregation trees in wireless sensor networks that minimize the number of redundant messages between transmitters and collectors.

In [21], message bundles are used to reduce the number of messages required by a parallel algorithm to communicate data and task information. Messages are bundled according to their importance in the context of the algorithm.

A control message bundling framework was proposed in [22] for multicast environments and hierarchical protocols. The aggregation mechanism is based on the maximum number of messages and the timeout limit of each message in the packet. Contrarily to our work, application messages are not combined into packets and the impact of aggregation on trees with known structure is presented as an open issue.

Finally, there are other broadcast algorithms based on spanning trees built on hypercube-like topologies [5], [23], [24]. The main difference is that they assume a physical hypercube deployed in hardware on top of which the algorithms run.

3. System Model

We consider a distributed system that consists of a finite set P of $n > 1$ processes $\{p_0, .., p_{n-1}\}$ that communicate only by message passing. Processes are organised in a logical hypercube of dimension $d = \log_2 n$. Each single process executes one task and runs on a single processor. Therefore, the terms node and process are used interchangeably in this work.

Processes communicate by sending and receiving messages using atomic point-to-point primitives. The network is fully connected, but processes are organized on a virtual hypercube-like topology, called VCube (Section 3.1). Processes can fail by crashing and there is no recovery. If a process never crashes during a run, it is considered *correct* or *fault-free*; otherwise it is considered to be *faulty*. After any crash, the topology changes, but VCube maintains its logarithmic properties. There is no network partitioning.

Links are reliable, and, thus, messages exchanged between any two correct processes are never lost, corrupted or duplicated.

The system is synchronous, i.e., relative processor speeds and message transmission delay are bounded. In this scenario, VCube implements a perfect failure detector, i.e., faults are detected within a finite amount of time and there is no false suspicion.

3.1. The VCube Topology

VCube [10] is a distributed diagnosis algorithm that organizes the correct processes of the system P on a virtual hypercube-like topology. In a hypercube of d dimensions, called d-VCube, there are $n = 2^d$ processes. From the point of view of process i the other $n-1$ processes are grouped in $\log_2 n$ clusters, denoted by $c_{i,s}$, where $s = 1, ..., \log_2 n$ is the cluster index; cluster $c_{i,s}$ has size 2^{s-1}. The ordered set of processes in each cluster can be computed with the function below, in which $\oplus$ denotes the bitwise exclusive *or* operator (xor).

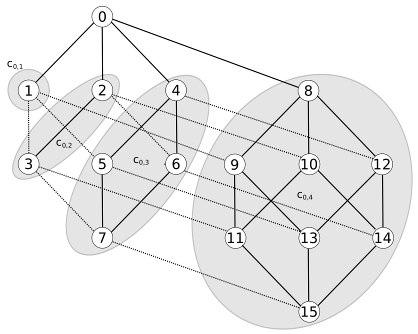
$$c_{i,s} = i \oplus 2^{s-1} \parallel c_{i \oplus 2^{s-1},k} \mid k = 1, .., s-1 \qquad (1)$$

Process i executes tests on other processes in $c_{i,s}$ to check whether it is correct or faulty. The test consists of comprehensive procedure that allows the tester to confirm the state of the tested node. If a correct reply is received within an expected time interval, the monitored process is considered to be alive (correct). Otherwise, it is considered to be faulty. In terms of the properties proposed by Chandra and Toueg [25] for unreliable failure detectors, VCube ensures the *strong completeness* property (eventually every process that crashes is permanently suspected by every correct process) and *strong accuracy* (correct processes are never suspected to have failed, i.e., no false suspicious).

Figure 1 shows an example of the hierarchical organization of process p_0 in a hypercube with four dimensions ($n = 2^4$ processes). Each cluster is presented in the table. For example, in the first round, process p_0 tests the first process (p_1) in the cluster $c_{0,1} = (1)$ and obtains information about the state of the other processes stored in p_1. Then p_0 tests the process p_2, which is the first process in the cluster $c_{0,2} = (2, 3)$. Next, p_0 runs tests in process p_4 of the cluster $c_{0,3} = (4, 5, 6, 7)$. Finally, p_8 is tested by p_0 in cluster $c_{0,4} = (8, 9, .., 15)$. As each process performs tests concurrently at the end of the last round each process will be tested at least once by another process. This ensures that an event is detected by all correct processes in $\log_2^2 n$ rounds.

Nodes keep state information as timestamps, which are state counters initialized as zero and incremented every time a state change is detected. If tests are represented by arcs from tester to tested node, each process i connects itself to one fault-free process of each cluster s, if there is one. If there are no faults, a complete logical hypercube is created.

In order to avoid that several processes test the same processes in a given cluster, process i executes a test on process $j \in c_{i,s}$ only if process i is the first faulty-free



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	
4	8 ..	9 ..	10 ..	11 ..	12 ..	13 ..	14 ..	15 ..	

Figure 1. 4-VCube hierarchical organization.

process in $c_{j,s}$. Thus, any process (faulty or fault-free) is tested at most once per round. The average latency can be reduced if every time a fault-free node is tested the tester and tested node exchance new information that was not available the last time they communicated.

4. The Spanning Tree Algorithm

Let $G = (V, E)$ be a graph that represents a distributed system with $|V|$ nodes and $|E|$ links. We consider that G consists of a single connected component. A spanning tree is defined as a connected and acyclic graph that contains all vertex of G [26].

Based on the VCube topology, We define the following two functions:

- $cluster_i(j) = s$: Let i and j be two nodes of the system, $i \neq j$. Function $cluster_i(j) = s$ returns the index s of the cluster of node i that contains node j, $1 \leq s \leq \log_2 N$. For instance, in the 4-VCube shown in Figure 1, $cluster_0(1) = 1$, $cluster_0(2) = cluster_0(3) = 2$, $cluster_0(4) = 3$, and $cluster_0(8) = 4$. Note that for any i, j, $cluster_i(j) = cluster_j(i)$.

- $FF\_neighbor_i(s) = j$ (*FF=Fault-Free*), returns the first correct process j in cluster s of process i, i.e., j is the first correct process of the list generated by the $c_{i,s}$. For example, in the table of Figure 1, $FF\_neighbor_0(1) = 1$, $FF\_neighbor_0(2) = 2$, $FF\_neighbor_0(3) = 4$ and $FF\_neighbor_0(4) = 8$. If process 2 is faulty, $FF\_neighbor_0(2) = 3$.

We then define function $subtree_i(j)$ as the set of neighbors of process i related to a source process j :

$$subtree_i(j) = \{\forall k = FF\_neighbor_i(s)\}$$

$$s = \begin{cases} 1..\log_2 n, \text{if } i = j \\ 1..(cluster_i(j) - 1), \text{otherwise} \end{cases} \quad (2)$$

Let $correct_i$ be the set of nodes that node i considers to be correct, based on the tests executed and information received by node i through the VCube.

Using the above functions, the following algorithm, proposed in [8], creates a spanning tree, rooted at i, propagating m over all non faulty nodes of the system:

Algorithm 1 Spanning tree algorithm at process i

```
1: $correct_i \leftarrow \{0, .., n-1\}$

2: procedure StartTree( )
3:     for all $k \in subtree_i(j)$ do
4:         Send($\langle m \rangle$) to $p_k$

5: upon Receive $\langle m \rangle$ from $p_j$
6:     if $j \in correct_i$ then
7:         for $k \in subtree_i(cluster_i(j) - 1)$ do
8:             send($\langle m \rangle$) to $p_k$

9: upon notifying crash($j$)
10:    $correct_i \leftarrow correct_i \setminus \{j\}$
11:    if $\exists k = FF\_neighbor_i(cluster_i(j))$ then
12:        send($\langle m \rangle$) to $p_k$
```

The set of children of i, the root of the spanning tree, are given by $subtree_i(\log_2 n)$. Whenever a node k receives m from its parent p, k becomes the root of a subtree with $s = cluster_k(p) - 1$ children. It identifies its non faulty children by calling $subtree_k(p)$ and then forwards m to them. The latter behave similarly, if they do not fail neither are leaves. Node k is a leaf if either $cluster_k(p) = 1$ or there is no correct processes in $c_{p,s}$, $s = cluster_p(k) - 1$.

All correct processes learn about the state of the other processes through VCube. If a crash is detected, say j, all processes execute the event Crash(j) (line 9). Process j, is removed from the local list of correct processes and m is sent to the next correct process k in the same cluster that contains j (if one exists).

Actually, a VCube spanning tree creates a binomial tree [27]. A binomial tree of order 0 is a single node (2^0). The binomial spanning tree of order d has 2^d nodes and it is recursively built from a root node whose children are roots of binomial trees of orders $d-1$, $d-2$, .., 1, 0. In fact, each child of a process i is the root of a subtree of the cluster in which it is the first correct process according the $c_{i,s}$. Note that, if there exist faulty processes, a incomplete binomial tree is built.

For example, considering a fault-free scenario, the spanning tree of process 0 in Figure 1 is build by sending the message to the first correct processes in each cluster $s = 1, .., \log_2 n$, represented by the processes 1, 2, 4 and 8. When these process receives the message, they become the root of the subtree in their own cluster. Figure 2(a) shows the full tree of process 0.

4.1. Message Bundling for Overlapping Trees

Message bundling is a way to optimize resource usage in distributed systems by reducing the number of messages sent, and decreasing the overhead of transport and routing protocols. The trees in Figure 2 show the potential of bundling broadcast messages that are sent on multiple trees built on a VCube. Comparing the trees of p_0 and p_4, for example, it is possible to see that many small subtrees are overlapped. If we compare p_0 and p_8's trees, we note that only the edge between the two processes themselves is not shared. In this case, a message sent by p_0 can be bundled with a message sent by p_8, and a single message can be sent on a subtree with half the number of nodes in the system.

5. The Hierarchical Broadcast Algorithm

Best-effort broadcast ensures the deliver of the messages from a correct source to all correct processes in the system, contrarily to the reliable broadcast which guarantees message delivery even if the source fails.

An algorithm of best-effort broadcast should ensure the *Validity* and *Integrity* properties. Validity states that if a correct process broadcasts a message m, then it eventually delivers m; and Integrity guarantees that every correct process delivers the same message at most once (no duplication) and only if that message was previously broadcast by some process (no creation).

Several broadcast algorithms use spanning trees to propagate messages since they are scalable and avoid the ack implosion problem. In this section we present a best-effort broadcast algorithm which exploits spanning tree built over the VCube topology as well as a bundling message strategy.

5.1. The Best-Effort Broadcast Algorithm

Algorithm 2 presents our hierarchical best-effort broadcast algorithm with message bundling. It dynamically builds spanning trees on VCube which also provide information about the state (correct or faulty) of the processes of the system. The algorithm tolerates up to $n-1$ process crashes.

Each message m from the application have three attributes: (1) the identification of the source process, i.e., the processes that starts the broadcast, represented by $m.src$; (2) the *timestamp*, a message counter $m.ts$ that uniquely identifies each message broadcast by the same source; and (3) the size of the message ($m.size$).

The algorithm uses two types of messages:

- $\langle TREE, m\rangle$: identifies the application message m being propagated;
- $\langle ACK, (src, ts)\rangle$: confirms the receipt of m by the destination, $src = m.src$ and $ts = m.ts$.

Two parameters are required by the algorithm: $maxDelay$, which is the maximum delay that a bundled message can be held by a node before forwarding; and $maxPayload$, which corresponds to the maximum packet size.

The local variables kept by process i are:

- $correct_i$: set of processes considered to be correct by process i;
- $rcv\_base_i[n]$: keeps, for each process j, the *timestamp* of a message m broadcast by j and delivered by i such that all messages broadcast by j before m were also delivered by i. We denote m the last message delivered in order.
- $rcv\_win_i[n]$: for each processes j, it keeps the list of messages delivered by i whose timestamp is greater than $rcv\_base_i[j]$, i.e., if $m \in rcv\_win_i[j]$ not all messages whose timestamp is smaller then $m.ts$ were delivered by i.
- ack_set_i: the set with all pending acks for process i. For each message $\langle TREE, m\rangle$ received by process i from a process j and forward to process k, an element $\langle j, k, (m.src, m.ts)\rangle$ is added to this set;
- msg_set_i: set of messages with pending ack at process i, used for retransmissions in case of faults;
- $delay\_msg_i[n]$: list of bundled messages waiting to be sent to the same destination process;
- $timer_i[n]$: for each neighbor j, $timer_i[j]$ controls the remaining time that i can keep a bundled message to be sent to j. When $timer_i[j]$ expires, i must forward the bundled message to j. The function STOP $timer_i[j]$ turns off the timer related to process j and the function START $timer_i[j]$ starts the timer of process j with the value of the parameter $maxDelay$.

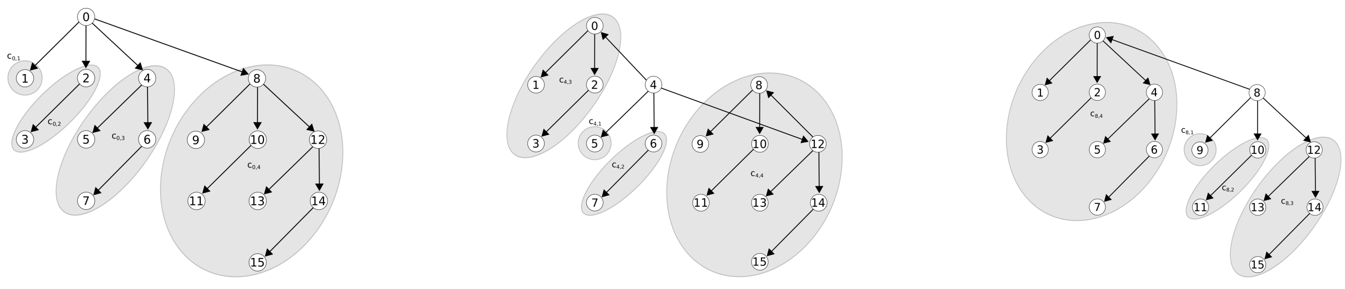
The asterisk is used as a wildcard to select ACKs in the set ack_set. An element $\langle j, *, m\rangle$, for example, represents all pending acks for a message m received by process j and forward to any other process.

Source process i broadcasts message m by invoking the BROADCAST procedure, which calls HANDLE_MESSAGE to handle m, with $m.src$ identifying the source process and $m.ts$ the message timestamp. The total message size is computed using the $length(m)$ function based on the application data in the message ($m.size$) plus fields introduced by the broadcast layer.

In order to be able, in case of failure, to retransmit a previously sent message, every message $\langle TREE, m\rangle$ sent by i to its neighbors is temporarily kept in msg_set_i (line 11) while there exist pending *ack*s for this message (line 84). Then, line 13, using rcv_base_i and rcv_win_i variables, verifies if m has not already been delivered by i, ensuring, therefore, the no duplication integrity property. If m is the first message received by i from $m.src$, i.e., $rcv\_base_i[m.src] = \perp$ or the messages has not been delivered yet (if it is the case, $m.ts$ will be either smaller than $rcv\_base_i[m.src]$ or $m \in rcv\_base_i[m.src]$), m is delivered (line 14) and the procedure UPDATE_WINDOW is called (line 15) for updating rcv_base_i and rcv_win_i in order to guarantee that i will not deliver m again (Integrity property). Being m the first message received by i from $m.src$, if m is also the first message broadcast by $m.src$

Algorithm 2 Best-Effort Broadcast with Bundling Messages at process p_i

```
Require: $maxDelay$                                    ▷ Maximum time a message can be delayed
Require: $maxPayload$                                  ▷ The maximum size of a set of bundled messages

 1: upon Initialization
 2:     $ack\_set_i \leftarrow \emptyset$                                     ▷ Set of pending acks
 3:     $msg\_set_i \leftarrow \emptyset$                                     ▷ Set of messages with pending acks
 4:     $correct_i = \{0, .., n-1\}$                                          ▷ Set of correct processes
 5:     $\forall j \in 0..n-1 : rcv\_base_i[j] \leftarrow \perp$               ▷ $ts$ of the last message from $j$ delivered in order
 6:     $\forall j \in 0..n-1 : rcv\_win_i[j] \leftarrow \emptyset$            ▷ $ts$ of the messages from $j$ delivered out of order
 7:     $\forall j \in FF\_neighbor_i(s) \mid s = 1..\log_2 n : delay\_msg_i[j] = \emptyset$, $timer_i[j]$ is turned off

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

10: procedure HANDLE_MSG(message $\langle TREE, m \rangle$, process $j$)
11:     $msg\_set_i \leftarrow msg\_set_i \cup \{m\}$
12:     if $rcv\_base_i[m.src] = \perp$ or ($m.ts > rcv\_base_i[m.src]$
13:             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(message $m$)
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 $\exists\ m' \in msg\_win[m.src] \mid$
29:             $m'.ts = rcv\_base_i[m.src] + 1$ do
30:         $rcv\_base_i[m.src] \leftarrow m'.ts$
31:         $rcv\_win_i[m.src] \leftarrow rcv\_win_i[m.src] \setminus \{m'\}$

                                            ▷ Forward $m$ to all fault-free neighbors
32: procedure FORWARD(message $\langle TREE, m \rangle$, process $j$)
33:     for all $k \in subtree_i(m.src)$ do
34:         CHECK_BUNDLE($\langle TREE, m \rangle$, $k$)
35:         $ack\_set_i \leftarrow ack\_set_i \cup \{\langle j, k, (m.src, m.ts) \rangle\}$

36: procedure CHECK_BUNDLE(message $\langle T, c \rangle$, process $k$)
37:     if $length(\langle T, c \rangle) \geq maxPayload$ then
38:         SEND($\langle T, c \rangle$) to $p_k$
39:     else if $length(delay\_msg_i[k] \cup \langle T, c \rangle) > maxPayload$
    then
40:         SEND($delay\_msg_i[k]$) to $p_k$
41:         $delay\_msg_i[k] \leftarrow \{\langle T, c \rangle\}$
42:         stop $timer_i[k]$
43:     else if $length(delay\_msg_i[k] \cup \langle T, c \rangle) = maxPayload$
    then
44:         SEND($delay\_msg_i[k] \cup \{\langle T, c \rangle\}$) to $p_k$
45:         $delay\_msg_i[k] \leftarrow \emptyset$
46:         stop $timer_i[k]$
47:     else
48:         $delay\_msg_i[k] \leftarrow delay\_msg_i[k] \cup \{\langle T, c \rangle\}$
49:     if $timer_i[k]$ is off and $delay\_msg_i[k] \neq \emptyset$ then
50:         start $timer_i[k]$

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

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

62: procedure RECEIVE(message $\langle TREE, m \rangle$, process $j$)
63:     if $\{m.src, j\} \nsubseteq correct_i$ then
64:         return
65:     HANDLE_MSG($\langle TREE, m \rangle$, $j$)

66: procedure RECEIVE(message $\langle ACK, (src, ts) \rangle$, process $j$)
67:     $k \leftarrow x : \langle x, j, (src, ts) \rangle \in ack\_set_i$
68:     $ack\_set_i \leftarrow ack\_set_i \setminus \{\langle k, j, (src, ts) \rangle\}$
69:     CHECK_ACKS($(src, ts)$, $k$)

70: upon $timer_i[k]$ expiration
71:     SEND($delay\_msg_i[k]$) to $p_k$
72:     $delay\_msg_i[k] \leftarrow \emptyset$

73: upon notifying CRASH(process $j$)          ▷ $j$ is detected as faulty
74:     $correct_i \leftarrow correct_i \setminus \{j\}$
75:     $delay\_msg_i[j] \leftarrow \emptyset$; stop $timer_i[j]$
76:     $k \leftarrow FF\_neighbor_i(cluster_i(j))$
77:     for all $p = x, q = y, (src, ts) = z : \langle x, y, z \rangle \in ack\_set_i$ do
78:         if $\{src, p\} \nsubseteq correct_i$ then
79:                                                   ▷ Remove pending acks
80:             $ack\_set_i \leftarrow ack\_set_i \setminus \{\langle p, q, (src, ts) \rangle\}$
81:         else if $q = j$ then
                              ▷ Send $m$ to the new correct neighbor $k$, if exists
82:             if $k \neq \perp$ and $\langle p, k, (src, ts) \rangle \notin ack\_set_i$ then
83:                 $ack\_set_i \leftarrow ack\_set_i \cup \{\langle p, k, (src, ts) \rangle\}$
84:                 $m \leftarrow m' \in msg\_set_i : m'.src = src, m'.ts =$
    $ts$
85:                 CHECK_BUNDLE($\langle TREE, m \rangle$, $k$)
86:             $ack\_set_i \leftarrow ack\_set_i \setminus \{\langle p, j, (src, ts) \rangle\}$
87:         CHECK_ACKS($(src, ts)$, $p$)
```



(a) Tree of process p_0 (b) Tree of process p_4 (c) Tree of process p_8

Figure 2. Examples of spanning trees in a 4-VCube.

($m.ts = 0$), $rcv\_base_i[m.src]$ is updated with $m.ts = 0$, otherwise m is included in $rcv\_win_i[m.src]$. In this way, i will not deliver m again. On the other hand, if m is not the first message received by i broadcast by $m.src$, it first checks if m was the next message broadcast by $m.src$ just after the last message delivered in order by i, i.e., if $m.ts = rcv\_base_i[j] + 1$. If this is the case, $rcv\_base_i[m.src]$ is updated with $m.ts$. Otherwise, m is included in $rcv\_win_i[m.src]$. The delivery of m might also entail the filling of some gaps of the current order of delivered messages. Hence, in lines 29-31 $rcv\_base_i[m.src]$ is updated with the timestamp value of the last message m' delivered in order by i that belongs to $rcv\_win_i[m.src]$ and all delivered messages whose timestamp is smaller or equal to $m'.ts$ are removed from $rcv\_win_i[m.src]$. By calling the procedure FORWARD, message m is either forwarded to to every correct neighbor k or kept in a buffer to be latter forwarded to k (CHECK_BUNDLE). Then, for each k, an *ack* $\langle j, k, (m.src, m.ts)\rangle$ is included in the set of pending *acks* (ack_set_i). If i has no correct neighbor or it is a leaf of the spanning tree ($cluster_i(j) = 1$), no *ack* is included in ack_set_i and the procedure is complete and i sends an ACK message to j by calling function CHECK_ACKS (line 17).

Procedure CHECK_BUNDLE is responsible for bundling and forwarding $TREE$ and ACK messages of process i to process k. If the size of the message is equal or greater than the maximum size of the network packet, denoted as $maxPayload$, the message is immediately sent (line 37). If the size of message m plus the size of those messages to k that have been delayed and kept in $delay\_msg_i[k]$ is greater than (resp. equal to) $maxPayload$, the messages in $delay\_msg_i[k]$ (resp. $delay\_msg_i[k]$ and m) are sent to k, $delay\_msg_i[k]$ will keep just m (resp. no message) and the timer $timer_i[k]$ is stopped. Otherwise, m is included in $delay\_msg_i[k]$. Then if the timer $timer_i[k]$ has been stopped, it will be restarted to $maxDelay$. Upon the expiration of $timer_i[k]$ (line 70), the messages in $delay\_msg_i[k]$ are sent within a single packet to k.

When receiving message m of type $TREE$ from process j (line 62), i will handle the message by calling HANDLE_MESSAGE, provided that the source of m ($m.src$) and j are both correct. If it is not the case, the reception is aborted since if j is faulty, the process that sent m to j, when detecting j's crash will retransmit the message. Thus j and i will receive m through a new spanning tree that does not contain j. If $m.src$ is faulty, it is not necessary to continue forwarding m since best effort broadcast does not guarantee that all correct processes will deliver m if $m.src$ has failed. On the other hand, when a message $\langle ACK, (src, ts)\rangle$ is received by i, the set ack_set_i is updated and, if there are no more pending *acks* related to $m : m.src = src, m.ts = ts$, CHECK_ACKS removes m from the set msg_set_i. If the process k, which has sent m to i, is not i itself and both the source of m and k are correct, then an ACK message is sent to k. If $k = i$, the latter is the source of m and the broadcast terminates.

Upon the detection that node j has crashed (CRASH event), (1) i updates the list of correct processes, (2) removes from the set ack_set all the *ack*s which have process j as the destination of or those that have j as the source of a message, and (3) retransmits to the first neighbor k of i in the same cluster of j, (if such a k exists) those messages sent to j that have not been acknowledge by it. These retransmissions to k will lead to other retransmissions over the newly rebuilt spanning tree.

The size of a message depends on its type. A $TREE$ message contains, besides the data from the application (APP), a header H which is composed by the type of the message *type* as well as the identifier of the source process (src) and the timestamp of the message ts, as previously described. On the other hand, an ACK message contains just the header H composed by the type *type* and the necessary information to identify the message which is acknowledged, i.e. (src, ts). The difference of the sizes of the two types of messages is basically the size of the application message. Therefore, we have:

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

Note that the size of a bundled message is given by the sum of the size of messages $TREE$ and ACK that it carries.

6. Performance Results

In this section we present the results of the simulation experiments comparing the proposed best-effort broadcast algorithm using bundling messages with the same algorithm without bundling. The algorithms were implemented using Neko [28][1]. The tests are divided in two parts. First the results are presented for fault-free scenarios, and then for scenarios with faulty processes. The use of multiple shared trees is presented in Section 6.2.

6.1. Simulation Parameters

For a process to send a message to more than one destination it executes primitive SEND sequentially to each destination in turn. For each message, t_s time units are used to the send the message at the source and t_r units to receive the message at the destination. The transmission delay t_t refers to the time it takes to transmit the message on each communication channel.

To evaluate the performance of broadcast solutions, three metrics are used: (1) the latency to deliver the broadcast message to all the correct processes; (2) the total number of messages sent by the algorithm, including acknowledgments; and (3) the message sizes, in bytes. In each experiment a single message is broadcast at the beginning of the simulation.

The proposed algorithm was evaluated in scenarios with different numbers of processes and, in faulty scenarios, with different numbers of faulty processes. The time intervals were set as $t_s = t_r = 0.1$ and $t_t = 0.8$ time units. The testing interval of the VCube is 30.0 units of time. In addition, a process is considered to be faulty if a reply does not arrive at the tester after $4 * (t_s + t_r + t_t)$ time units.

Five different scenarios were generated by varying the following parameters: (1) the maximum packet size of bundled messages; (2) the maximum delay for sending a packet; (3) the size of $TREE$ messages; and (4) the size of ACK messages. The parameters used in each scenario are shown in Table 1, in which each column represents one of the scenarios.

In the scenario (NO-AGGR), no message is bundled. To avoid changes to the algorithm, the maximum packet and message size of $TREE$ and ACK has been set to 1, and the maximum delay is zero. In the other scenarios, the sizes of the messages and the delay for sending messages varied. In SMALL scenarios, the size of $TREE$ is 24, slightly larger than that of ACK which is 20, representing applications that propagate little information, e.g. a value reported by a sensor. For BIG scenarios, the ACK size remains 20 and the $TREE$ messages have size 500. The maximum delay was set to 2 or 10 time units and this is identified by the label of the scenario. In the SMALL2 scenario, for example, the maximum packet size is 1460, the $TREE$ messages have size 24, and the maximum delay is 2. In the BIG10 scenario, the maximum packet size is also 1460, the $TREE$ messages have size 500, and the maximum delay is 10. For comparison purposes, the NO-AGGR scenario assumes message sizes equivalent to the compared scenario.

[1]The complete source code and configuration files are available at www.inf.unioeste.br/~luiz

TABLE 1. SCENARIO CONFIGURATION PARAMETERS

	NO-AGGR	SMALL2	BIG2	SMALL10	BIG10
Maximum packet length	1	1460	1460	1460	1460
TREE message length	1	24	500	24	500
ACK message length	1	20	20	20	20
Maximum delay	0	2	2	10	10

6.2. The Impact of Multiple Spanning Trees

In order to evaluate the proposed strategy on extreme scenarios we executed an experiment in which 16 processes broadcast messages on a varying number of trees. Each process broadcasts a single message using a fixed set of trees, for a total of 16 messages. Initially, a single tree is used by all processes, in this case all trees totally overlap, but the contention for using that tree increases. Then two trees are used, and so on until each process uses its own tree. The trees used in each scenario are employed processes with identifiers that are smaller than the number of trees, i. e., with one tree, p_0's tree is used to send 16 messages; with two trees, p_0 and p_1's trees are used, each one sending 8 messages. We only present scenarios in which the number of messages is divisible by 16 (the number of processes).

In Table 2 we present the latency and number of messages for each scenario, and the multiple trees employed. Results show that in the non-aggregation scenario (NO-AGGR) the latency decreases when multiple trees are used, reaching a reduction of more than 100% when each process uses its own tree. This is due to the contention for the root whenever a tree is shared by multiple processes, which is especially high in the scenario with a single tree, in which p_0's tree is used by *all* the processes. Remember that in the proposed model, for each message sent there is a cost in terms of the processing time. The number of messages (TREE and ACK) of the non-aggregation scenarios is constant and equal to $n(2n-2)$, since there are no failures. In scenarios with bundled messages the latency includes the delays for aggregating messages and the propagation in the tree. However, there is an advantage in sharing the trees in terms of the number of messages required, since the trees overlap multiple times.

6.3. Fault-Free Scenarios

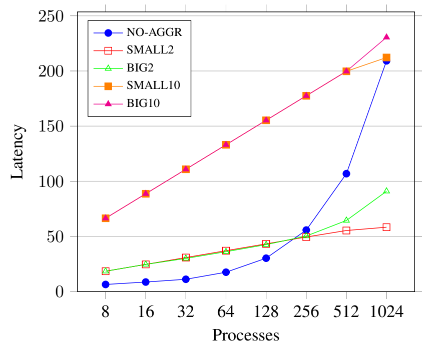
Figure 3 shows the results obtained for systems of different sizes and without faulty processes. The latency represents the total time to complete the broadcast for all the processes in the five proposed scenarios, that is, the total simulation time considering that all processes execute a single broadcast. The number of messages shown was

TABLE 2. LATENCY (LAT.) AND NUMBER OF MESSAGES (#MSG) USING MULTIPLE TREES WITH 16 PROCESSES

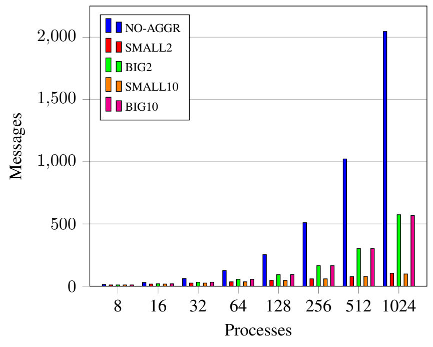
	NO-AGGR		SMALL2		BIG2		SMALL10		BIG10	
Trees	Lat.	#msg	Lat.	#msg	Lat.	#msg	Lat.	#msg	Lat.	#msg
1	14.7	480	24,7	30	25,5	168	88.7	30	88.7	162
2	11.6	480	24,7	60	24,7	188	88.7	60	88,7	190
4	9.0	480	24,7	104	24,7	192	88.7	104	88.7	192
8	7.2	480	24,7	176	24,7	192	88.7	176	88.7	192
16	6.1	480	24,7	272	24,7	304	88.7	272	88.7	304

computed as the average number of messages sent by each process.

The scenario with no bundling presents lower latency in comparison with the scenarios with bundling for systems with fewer than 256 processes, although the number of messages is much higher. In scenario SMALL2 the application messages are small the latency and number of messages are the best of all scenarios. Also noticeable is that although SMALL10 presents the same latency as BIG10, it employed a significantly smaller number of messages. Note that the delays for sending the messages have a similar impact on the latency in the scenarios with bundling, regardless of the number of messages bundled per packet. However, it appears that the impact tends to decrease as the load increasing in systems with smaller messages systems (SMALL), as happened for the 1024 process scenarios.



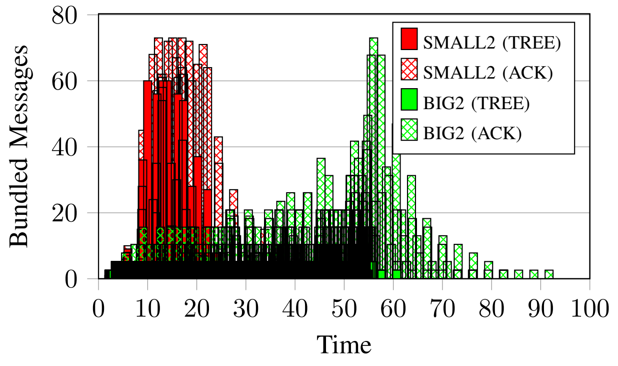
(a) Latency



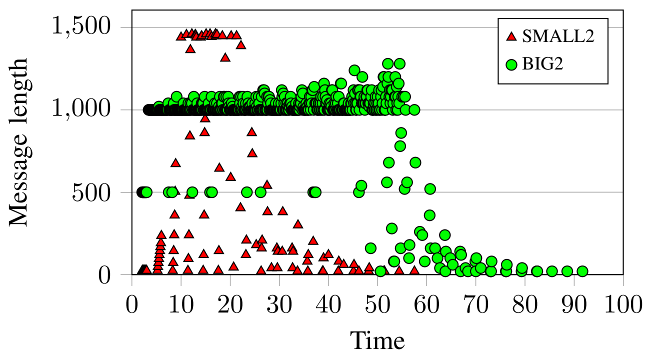
(b) Messages

Figure 3. Latency and average number of messages to broadcast one message per process in a fault-free execution.

Figure 4(a) shows a comparison of the performance of the fault-free SMALL2 and BIG2 scenarios, in which p_0 executes broadcast with bundling in a system with 1,024 processes. Due to the smaller size of SMALL2 application messages, the number of messages that can be bundled in a single packet is higher, especially at the beginning of the simulation. For BIG2, blundling has a greater impact at the end of the simulation, as ACKs have smaller size.



(a) Bundled Messages



(b) Message Length

Figure 4. Number of bundled messages and message sizes at process p_0 in the SMALL2 and BIG2 scenarios with 1024 processes (fault-free execution).

Still comparing the 1,024 processes for SMALL2 and BIG2 in terms of the size of the messages sent by the process p_0, it can be observed in Figure 4(b) that SMALL2 is able to take better advantage of bundling given the packet size (1460 bytes) as the number of messages increases. However, BIG2 still shows good performance especially given ACK bundling.

In terms of the total of bytes transmitted, we compared SMALL2 and BIG2 with NO-AGGR scenarios. In fault-free systems with 1,024 processes, 2,046 messages per broadcast are generated (1,023 TREEs and 1,023 ACKs). Comparing SMALL2 with NO-AGGR using the message size parameters of Table 1 and a network with standard header of 40 bytes, this generates a header overhead of 2,095,104 * 40 = 80 Mb and 106,496 * 40 = 4 Mb, respectively. Note that the difference is reduced due to the effectiness of bundling in NO-AGGR with larger messages, which confirms the advantage of bundling for applications that exchange small messages frequently, as in the SMALL scenarios.

6.4. Faulty Scenarios

To evaluate the impact of failures on the proposed solution, experiments were performed simulating process crashes. Two types of experiments were performed. In the first, a single process fails at the beginning of the simulation; in the second experiment failures were generated randomly during the simulation.

Fault of a single process. Initially process p_1 was set to fail at time $t = 0.0$, i.e., at the beginning of the simulation. The crash is detected by p_1's VCube neighbors (p_0, p_3 and p_5) and propagated according to the VCube strategy. Each process, upon detecting the crash of p_1, performs the steps indicated by the CRASH event of the algorithm. New $TREE$ messages are transmitted as the VCube reconfigures itself.

Table 4 compares the average number of messages (TREE and ACK) transmitted by each correct process in a fault-free execution (FF) and with faulty process p_1 (FY). In the scenario without bundling, each broadcast requires with one less message, i.e. the ACK that is not returned by the faulty process. Note that in the beginning of the simulation the processes do not yet know that p_1 is faulty and, therefore, the $TREE$ messages are sent to it normally. The VCube detection latency causes an increase of the execution time in the faulty scenarios as shown in Table 3. Although it has an impact on all scenarios, VCube's latency is most evident in scenarios without bundling (NO-AGGR) or with small messages (SMALL2 and BIG2). In the SMALL10 and BGI10 scenarios the aggregation delay dominates the fault detection latency.

Multiple Faults. Finally, multiple crashes were injected in the scenarios, where the number of faulty processes was equal to $(log_2 n) - 1$. The faulty processes and the time at which the failures were generated were random. Each scenario was run with 10 different fault configurations. Table 5 shows the latency obtained. The number of messages was omitted due to lack of space and because the time instant a process fails has a direct impact on the number of messages sent, what is not useful without an in-depth analysis of each case. We observed a greater variation in latency for systems with smaller number of processes, since in these systems, the tree propagation latency is lower and the crash detection time has more impact on the total time. The confidence interval (CI) shows the impact that failures have on systems with a higher proportion of failed processes, as in scenarios with a smaller number of processes.

7. Conclusion

This paper presented an approach to bundle messages along spanning tree intersections which are frequently employed by broadcast algorithms. We proposed a best-effort broadcast algorithm with message bundling. This is an autonomous algorithm builds and dynamically maintains spanning trees rotted at each source process are on top of a VCube, a scalable topology which adapts itself after process crashes. Simulation results comparing the proposed solution with a alternative that does not bundle messages show the efficiency of the proposed strategy in fault-free and faulty scenarios, especially in terms of the number and size of the messages required. Future work includes adapting the algorithm for the asynchronous model, evaluting the impact of bundling messages on other types of broadcast (e.g. causal) and also implementing a TCP/IP broadcast tool based on the algorithm.

Acknowledgment

This work was partially supported by grants Fundação Araucária/SETI 341/10 and CNPq-309143/2012-8.

References

[1] V. Hadzilacos and S. Toueg, "Fault-tolerant broadcasts and related problems," S. Mullender, Ed. New York, NY, USA: ACM Press, 1993, ch. Distributed systems, pp. 97–145.

[2] F. B. Schneider, D. Gries, and R. D. Schlichting, "Fault-tolerant broadcasts," *Sci. Comput. Program.*, vol. 4, no. 1, pp. 1–15, May 1984.

[3] P. Fragopoulou and S. Akl, "Edge-disjoint spanning trees on the star network with applications to fault tolerance," *IEEE Trans. Comput.*, vol. 45, no. 2, pp. 174 –185, Feb. 1996.

[4] K. Kim, S. Mehrotra, and N. Venkatasubramanian, "FaReCast: Fast, reliable application layer multicast for flash dissemination," in *ACM/IFIP/USENIX 11th Int'l Conf. on Middleware*, ser. Middleware'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 169–190.

[5] P. Ramanathan and K. Shin, "Reliable broadcast in hypercube multicomputers," *IEEE Trans. Comput.*, vol. 37, no. 12, pp. 1654–1657, 1988.

[6] M. Raynal, J. Stainer, J. Cao, and W. Wu, "A simple broadcast algorithm for recurrent dynamic systems," in *Proceedings of the 2014 IEEE 28th International Conference on Advanced Information Networking and Applications*, ser. AINA '14, 2014, pp. 933–939.

[7] L. A. Rodrigues, J. Cohen, L. Arantes, and E. P. Duarte, "A robust permission-based hierarchical distributed k-mutual exclusion algorithm," in *2013 IEEE 12th International Symposium on Parallel and Distributed Computing*, June 2013, pp. 151–158.

[8] L. A. Rodrigues, L. Arantes, and E. P. Duarte Jr., "An autonomic implementation of reliable broadcast based on dynamic spanning trees," in *10th European Dependable Computing Conference*, ser. EDCC'14, 2014, pp. 1–12.

[9] D. Jeanneau, L. A. Rodrigues, L. Arantes, and E. P. Duarte, "An autonomic hierarchical reliable broadcast protocol for asynchronous distributed systems with failure detector," in *7th Latin-American Symp. Dep. Comput. (LADC)*, Oct 2016, pp. 91–98.

[10] E. P. Duarte Jr., L. C. E. Bona, and V. K. Ruoso, "VCube: A provably scalable distributed diagnosis algorithm," in *5th Work. on Latest Advances in Scalable Algorithms for Large-Scale Systems*, ser. ScalA'14. Piscataway, USA: IEEE Press, 2014, pp. 17–22. [Online]. Available: http://dx.doi.org/10.1109/ScalA.2014.14

[11] A. Gupta, B. Liskov, and R. Rodrigues, "Efficient routing for peer-to-peer overlays," in *Proceedings of the 1st Conf. on Symp. on Networked Systems Design and Implementation*, ser. NSDI'04. Berkeley, CA, USA: USENIX Association, 2004.

[12] N. Hidalgo, L. Arantes, P. Sens, and X. X. Bonnaire, "An aggregation-based routing protocol for structured peer to peer overlay networks," in *AP2PS 2010 - 2nd Intl. Conf. on Advances in P2P Systems*, 2010, pp. 76–81.

[13] K. Shudo, "Message bundling on structured overlays," in *2017 IEEE Symp. on Computers and Communications (ISCC)*, July 2017, pp. 424–431.

TABLE 3. LATENCY TO FAULT-FREE (FF) AND ONE-FAULTY PROCESS (FY) SCENARIOS

Number of	NO-AGGR		SMALL2		BIG2		SMALL10		BIG10	
Processes	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	49.8	42.6	48.8	155.3	155.4	155.3	155.6
256	55.8	66.2	49.5	56.0	50.4	56.9	177.5	177.6	177.5	177.8
512	106.9	125.8	55.4	61.8	63.8	72.7	199.7	199.9	199.7	200.2
1,024	209.2	243.9	58.4	67.8	92.6	104.2	214.4	222.3	224.8	231.9

TABLE 4. NUMBER OF MESSAGES IN FAULT-FREE (FF) AND ONE-FAULTY PROCESS (FY) SCENARIOS

Number of	NO-AGGR		SMALL2		BIG2		SMALL10		BIG10	
Processes	FF	FY	FF	FY	FF	FY	FF	FY	FF	FY
8	112	91	80	67	80	67	80	59	80	59
16	480	435	272	247	304	273	272	235	304	256
32	1,984	1,889	800	766	1,024	981	800	749	1,024	952
64	8,064	7,860	2,240	2,191	3,584	3,429	2,240	2,167	3,520	3,349
128	32512	32,079	6,016	5,947	11,904	11,635	6,016	5,917	12,032	11,742
256	130,560	129,643	15,104	15,019	42,240	41,624	15,104	14,985	42,240	41,802
512	523,264	521,350	39,424	38,639	155,136	153,280	40,448	40,032	155,136	154,020
1,024	2,095,104	2,091,167	106,496	105,188	587,776	580,983	100,352	100,580	581,632	576,723

TABLE 5. LATENCY TO MULTIPLE FAULTS WITH 95% CONFIDENCE INTERVAL (CI)

Number of	NO-AGGR		SMALL2		BIG2		SMALL10		BIG10	
Processes	Avg.	CI	Avg.	CI	Avg.	CI	Avg.	CI	Avg.	CI
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
1,024	247.0	0.3	102.1	0.0	129.4	14.7	255.3	0.2	267.6	0.3

[14] L. Schmidt, N. Mitton, D. Simplot-Ryl, R. Dagher, and R. Quilez, "Dht-based distributed ale engine in rfid middleware," in *2011 IEEE Intl. Conf. on RFID-Technologies and Appl.*, Sept 2011, pp. 319–326.

[15] A. Koike, T. Ohba, and R. Ishibashi, "Iot network architecture using packet aggregation and disaggregation," in *Int'l Congress on Advanced Applied Informatics (IIAI-AAI)*, July 2016, pp. 1140–1145.

[16] R. Rajagopalan and P. K. Varshney, "Data-aggregation techniques in sensor networks: A survey," *IEEE Communications Surveys Tutorials*, vol. 8, no. 4, pp. 48–63, Fourth 2006.

[17] C. Weng, M. Li, and X. Lu, "Data aggregation with multiple spanning trees in wireless sensor networks," in *Int'l Conf. on Embedded Software and Systems*, Jul. 2008, pp. 355–362.

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

[19] M. Chetlur, N. Abu-Ghazaleh, R. Radhakrishnan, and P. A. Wilsey, "Optimizing communication in time-warp simulators," in *Parallel and Distributed Simulation, 1998. PADS 98. Proceedings. Twelfth Workshop on*, May 1998, pp. 64–71.

[20] L. A. Villas, D. L. Guidoni, R. B. Araújo, A. Boukerche, and A. A. Loureiro, "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*. New York, NY, USA: ACM, 2010, pp. 110–117.

[21] E. Reinhard and A. Chalmers, "Message handling in parallel radiance," in *18th European Conference on Parallel and Distributed Computing*, ser. Euro-Par 2012, Aug.

[22] S. Khanna, J. S. Naor, and D. Raz, "Control message aggregation in group communication protocols," in *Automata, Languages and Programming*, P. Widmayer, S. Eidenbenz, F. Triguero, R. Morales, R. Conejo, and M. Hennessy, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 135–146.

[23] J. Wu, "Optimal broadcasting in hypercubes with link faults using limited global information," *J. Syst. Archit.*, vol. 42, no. 5, pp. 367–380, 1996.

[24] J. Liebeherr and T. Beam, "HyperCast: A protocol for maintaining multicast group members in a logical hypercube topology," in *Networked Group Communication*, ser. LNCS, L. Rizzo and S. Fdida, Eds. Springer Berlin Heidelberg, 1999, vol. 1736, pp. 72–89.

[25] T. D. Chandra, V. Hadzilacos, and S. Toueg, "The weakest failure detector for solving consensus," vol. 43, no. 4, pp. 685–722, Jul. 1996. [Online]. Available: http://doi.acm.org/10.1145/234533.234549

[26] R. G. Gallager, P. A. Humblet, and P. M. Spira, "A distributed algorithm for minimum-weight spanning trees," *ACM Trans. Program. Lang. Syst.*, vol. 5, pp. 66–77, Jan. 1983.

[27] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*, 3rd ed. The MIT Press, 2009.

[28] P. Urbán, X. Défago, and A. Schiper, "Neko: A single environment to simulate and prototype distributed algorithms," *Journal of Inf. Science and Eng.*, vol. 18, no. 6, pp. 981–997, Nov. 2002.