A communication-efficient causal broadcast publish/subscribe system

João Paulo de Araujo

Thèse présentée pour obtenir le grade de docteur
Sorbonne Université

Laboratoire d'Informatique de Paris 6
École Doctorale Informatique, Télécommunications et Électronique (ED130)

Discipline : Informatique

A Communication-Efficient Causal Broadcast Publish/Subscribe System

---

Par : **João Paulo de Araujo**

Sous la direction de Pierre Sens

et encadrée par Luciana Arantes
et par Luiz Antonio Rodrigues

Rapporteurs : François Taïani

Matthieu Roy

Examinateurs : Béatrice Bérard

Christophe Cérin

Date de soutenance : le 05 avril 2019

To my mother, Teresa Aparecida Paulino.

Acknowledgements

First of all, I want thank Pierre Sens and Luciana Arantes for accepting to advise me and for the support since the very first day I arrived in France. I am very grateful for their guidance, assistance, and availability. Working close to them has been a very rewarding experience.

I would like to thank Béatrice Bérard, Christophe Cérin, Matthieu Roy, and François Taïani for accepting to be part of my thesis jury. Additionally I thank Matthieu Roy and François Taïani for accepting to review this manuscript.

I also thank Elias Procópio Duarte Jr. and Luiz Antonio Rodrigues. Their support has been of great importance for this research and I am grateful for the opportunity of having worked in collaboration with them.

Charlotte Lemasson deserves special thanks. Her patience, support, and caring were extremely important to me throughout this period. I am really pleased to have such a wonderful person by my side.

I could not forget to mention Gilson Miranda Jr. and Jaime Daniel Corrêa Mendes who, despite several thousand kilometers away, are friends that are always present just like in the good old times.

I also thank the colleagues at LIP6, specially the members of the DELYS team, for such a great environment, for the friendship, and everything else.

Finally, I gratefully thank the Brazilian National Council for Scientific and Technological Development (CNPq) for the scholarship they provided, which allowed me to undertake this research.

Abstract

Widely applied by many approaches, the Publish/Subscribe (Pub/Sub) paradigm enables nodes of a distributed system to disseminate information asynchronously. However, several existing Pub/Sub solutions present some limitations such as network contention of published messages or do not ensure causal order of delivered messages.

This thesis investigates how to provide a communication-efficient Pub/Sub system, that respects causal order of delivered messages within the same topic and copes with problems of traffic overhead and message contention that exist in several tree-based solutions. Contrarily to several existing tree-based broadcast protocols, the contributions presented in this thesis build distributed spanning trees on top of a hypercube-like topology, such that trees rooted at different nodes are differently organized.

The first contribution of the thesis consists of a causal broadcast protocol which reduces network traffic by aggregating messages without the use of timers. It exploits causal order of messages and common paths between different broadcast trees: the forwarding of some messages can be delayed and then combined with others, reducing message traffic. Different from existing timer-based approaches, it does not increase delivery latency.

The second contribution is a new topic-based Pub/Sub system, the *VCube-PS*, which ensures causal delivery order for messages published to the same topic. While several other tree-based Pub/Sub approaches use one tree per topic and *rendezvous* points, *VCube-PS* creates a new spanning tree rooted on the source of every message that is published. Such a per-publisher tree approach reduces node contention when compared to the single tree one, specially in scenarios with "hot topics", i.e., topics with high publication rates. Furthermore, the spanning tree built to disseminate a message associated to a given topic is composed only by known subscribers of the topic.

Both contributions were implemented on top of the event-driven simulator *PeerSim* and were assessed according to different metrics and scenarios. Results confirm that the proposed causal aggregation protocol reduces both network traffic and delivery latencies. Moreover, it also reduces the number of non-deliverable messages that are kept in nodes' buffers. Concerning the proposed Pub/Sub system, when compared to approaches with one single tree per topic, *VCube-PS* presented the lowest latency results when in presence of "hot topics", since it intrinsically provides load balancing of dissemination paths. Moreover, *VCube-PS* generates less message traffic than the former and the extra delay necessary to ensure causal delivery order represents only a small percentage of the end-to-end latency.

Keywords: Publish/Subscribe; Message Aggregation; Causal Broadcast; Hypercube Topology; Distributed Spanning Tree

Résumé

Largement utilisé par de nombreuses approches, le paradigme de Publication/Abonnement (*Publish/Subscribe*, Pub/Sub) permet aux nœuds d'un système distribué de diffuser des informations de manière asynchrone. Cependant plusieurs solutions de Pub/Sub existantes présentent certaines limitations, telles que la contention des messages publiés ou la non garantie de l'ordre causal des messages délivrés.

Cette thèse propose un système de Pub/Sub efficace, qui respecte l'ordre causal des messages publiés dans le même sujet (*topic*). Le système résout les problèmes d'overhead de trafic et de conflits de messages existant dans plusieurs solutions basées sur des arbres couvrants. Contrairement à plusieurs protocoles de diffusion existants basés sur des arbres, les contributions présentées dans cette thèse construisent des arbres couvrants répartis sur une topologie en hypercube, de telle sorte que les arbres avec différentes racines soient organisés différemment.

La première contribution de la thèse est un protocole de diffusion causale qui réduit le trafic réseau en agrégeant des messages sans utiliser de temporisateur. Ce protocole exploite l'ordre causal des messages et les chemins communs entre différents arbres couvrants pour agréger des messages. À la différence des approches existantes basées sur des temporisateurs, ce protocole n'augmente pas le temps d'attente avant la livraison des messages.

La deuxième contribution est un nouveau système de Pub/Sub basé sur des sujets, le *VCube-PS*, qui assure l'ordre de livraison causal des messages publiés sur le même sujet. Tandis que plusieurs autres approches utilisent un arbre par sujet, *VCube-PS* crée un nouvel arbre couvrant pour chaque message publié, où la racine de l'arbre est la source du message. Cette approche permet de réduire le problème de contention par rapport à des solutions avec un seul arbre par sujet, en particulier dans les scénarios comportant des "hot topics", c'est-à-dire des sujets avec des taux de publication élevés. De plus, l'arbre couvrant construit pour diffuser un message associé à un sujet donné est composé uniquement par les abonnés du sujet.

Les deux contributions ont été simulées avec PeerSim et ont été évaluées selon différents paramètres et scénarios. Les résultats confirment que le protocole d'agrégation causale proposé réduit à la fois le trafic réseau et les latences de livraison. En outre, cela réduit également le nombre de messages non livrables conservés dans les buffers des nœuds. En ce qui concerne le système de Pub/Sub proposé, *VCube-PS* présente les résultats de latence les plus faibles en présence de "hot topics" par rapport aux approches avec un seul arbre par sujet, car il fournit intrinsèquement une répartition de charge des chemins de diffusion. De plus, *VCube-PS* génère moins de trafic de messages et le délai supplémentaire nécessaire pour garantir l'ordre de livraison causal ne représente qu'un faible pourcentage de la latence.

Mots-clés: Publication/Abonnement; Agrégation de Messages; Diffusion Causale; Topologie en Hypercube; Arbre Couvrant Distribué

Contents

Chapter 1

Introduction

Contents

Asynchronous dissemination of information is a key feature in many recent distributed applications. The Publish/Subscribe (Pub/Sub) paradigm has emerged as a suitable middleware solution for this challenge due to its decoupling properties and scalability (Astley et al., 2004; Esposito et al., 2013).

In a Pub/Sub system, one or more *publisher* nodes produce messages that are consumed by *subscriber* nodes. Communication between these two types of participants is conducted using an overlay infrastructure, which ensures the delivery of published messages to all subscribers interested in those messages.

In order to receive publications, subscribers must inform the Pub/Sub system about its interests. Basically, there exist two Pub/Sub models, with respect to the way subscribers express their interests: topic-based and content-based. In the first one, nodes share a common knowledge about a set of topics and every message is labeled with one of these topics. Differently, in content-based systems, messages are classified according to attributes regarding the content of the message and subscribers express their interests by specifying constraints over the values of these attributes. Even if content-based Pub/Sub systems provide more flexibility for subscribers for defining their interests, the topic-based approach is exploited by a great number of applications such as notification frameworks, chat systems, distributed multi-player online games, among others. Commercial solutions (e.g., Firebase/Google Cloud Messaging, IBM MQ, Apache Kafka, etc.) also apply this model. Finally, online services such as Twitter can also be modeled as a topic-based Pub/Sub system.

This thesis addresses topic-based Pub/Sub systems by proposing a solution that efficiently

broadcasts publications to subscribers of a given topic by dynamically building spanning trees, rooted on the publisher and spread over those subscribers. Such a solution particularly focuses applications where some topics are highly popular ("hot topics").

Many existing topic-based Pub/Sub solutions organize subscribers of each topic in a tree, i.e., all messages of a given topic are broadcast using the same spanning tree, which can become a bottleneck (e.g., Scribe (Castro et al., 2002) and DYNATOPS (Zhao et al., 2013)). Furthermore, nodes which are not subscribers may also take part in the tree as relay nodes, increasing latency of message delivery. In applications that use topic-based Pub/Sub systems, it is known that publications are not evenly distributed among existing topics. For instance, in social networks like Twitter, most users tend to publish on a small number of topics (Sanli and Lambiotte, 2015). Thus, in Pub/Sub systems that use a unique per-topic tree to broadcast publications, highly demanded topics suffer from contention, due to the limited capacity of tree root nodes in dealing with publication frequency.

Many applications require that the delivery of publications to subscribers respect causal order of publication broadcast. In this case, if a node publishes a message after it has delivered another message, then no node delivers the latter after the former. For instance, if an online discussion system uses a topic-based Pub/Sub system in which each discussion group is represented by a topic, a question published on a group should never be delivered to any subscriber after an answer to that question which was also published in the same group, as the answer is causally related to the question. However, as far as our knowledge, despite its application usefulness, few existing Pub/Sub implement causal ordering of publications. Moreover, most of these works do not ensure causal order for messages published to the same topic (e.g., (Nakayama et al., 2016; Yamamoto and Hayashibara, 2017)). When provided (e.g., (Cugola et al., 2001)), the solution is not scalable because it induces message traffic due to extra acknowledgment messages.

1.1 Contributions

The first contribution of this thesis (Chapter 4) aims at providing a communication-efficient causal broadcast protocol that exploits causal order of messages and common paths between different broadcast trees: the forwarding of some messages can be delayed and then combined with others, without incurring any overhead and reducing message traffic. The second one (Chapter 5) presents a topic-based Pub/Sub system, *VCube-PS*, which ensures causal delivery order for messages published to the same topic and efficiently supports publication of messages to "hot topics", i.e., topics with high publication rates.

Contrarily to several existing tree-based broadcast protocols, the two contributions build spanning trees on top of *VCube* (Duarte et al., 2014), a diagnostic algorithm that organizes nodes of the system in a logical hypercube and presents logarithmic properties. Trees rooted on different nodes are differently organized.

Simulations for both contributions were implemented on top of the event-driven peer-to-peer simulator *PeerSim* (Montresor and Jelasity, 2009) and performance evaluation results are presented and discussed.

1.1.1 Causal Aggregation Broadcast

An issue concerning the performance of broadcast protocols is related to the amount of messages that are sent through the network. In Chetlur et al. (1998), the authors state that the cost of sending several small messages is higher than the cost of sending the same amount of data inside a single message.

Several existing approaches try to reduce message traffic by bundling messages into a single one. However, they usually apply timers for buffering messages, which increases end-to-end latencies.

This thesis proposes a tree-based causal broadcast protocol for bundling messages in which no timer is necessary. The protocol combines messages into a single one by taking advantage of the extra delivery delay that is imposed to a node when messages are received out of causal order. In other words, one of the criteria for bundling messages is based on the principle that, if a message is received before its causal dependencies at a node, the former will be necessarily delayed until the dependencies are received.

The algorithm for implementing the message aggregation approach relies on the inference rules of *VCube* (see Section 2.4) which allow a node to deduce, using only local information, every other node's spanning tree organization. Roughly, a node can delay the forwarding of a message to a child node in the spanning tree whenever it knows that this child has missing dependencies and it is also the responsible for forwarding the latter to this child node. Thus, based on the causal relation and path intersections, a node can decide to bundle causal related messages, without increasing end-to-end delivery latency.

Experimental results show that the proposed aggregation protocol reduces message traffic while not degrading delivery latency. In some high load scenarios, delivery latency is even reduced, due to the lower traffic-induced delay. Finally, average use of buffers for delayed messages also decreases because the number of nodes that can immediately deliver a message upon reception increases with the causal aggregation approach.

1.1.2 VCube-PS: A Topic-based Publish/Subscribe System

The new topic-based Pub/Sub system, *VCube-PS*, proposed in this thesis ensures that messages published to the same topic are delivered in causal order of their publications. While several existing approaches create a single tree per topic, *VCube-PS* dynamically creates, on top of *VCube*, trees rooted on the source of every published message.

By using per publisher spanning trees, *VCube-PS* alleviates the problem of contention that can occur in approaches where there exists only one single tree per topic (e.g., Scribe (Castro et al., 2002)). Thereby, it enables the efficient publication of high loads of publications to the same topic.

Differently from many existing approaches in which non subscribers may take part in a topic's publication diffusion tree, in *VCube-PS*, trees are built using only current subscribers. Those that unsubscribe from a topic will eventually not take part anymore in any spanning tree of this topic.

Evaluation results from simulation experiments confirm that *VCube-PS* performs better than single rooted tree-based Pub/Sub systems when there is a high publication rate per topic, since it provides load balancing. Moreover, decentralized broadcast of publications reduces delivery latencies. Finally, even if some publications are sent for a while to nodes that unsubscribe from the topic in question, such a scenario lasts temporarily and has an impact on the performance of only a small percentage of the overall number of publications.

1.2 Publications

The following articles were published during the development of this thesis.

1.2.1 Papers in International Conferences

- de Araujo, J. P., Arantes, L., Duarte, E. P., Rodrigues, L. A., and Sens, P. (2018). A communication-efficient causal broadcast protocol. In *Proceedings of the 47th International Conference on Parallel Processing*, ICPP 2018, pages 74:1–74:10, Eugène, OR, USA;

- Rodrigues, L. A., Duarte, E. P., de Araujo, J. P., Arantes, L., and Sens, P. (2018). Bundling messages to reduce the cost of tree-based broadcast algorithms. In *Proceedings of the 8th Latin-American Symposium on Dependable Computing*, LADC 2018, pages 115–124, Foz do Iguaçu, PR, Brazil;

- de Araujo, J. P., Arantes, L., Duarte, E. P., Rodrigues, L. A., and Sens, P. (2017). A publish/subscribe system using causal broadcast over dynamically built spanning trees. In *Proceedings of the 29th International Symposium on Computer Architecture and High Performance Computing*, SBAC-PAD 2017, pages 161–168, Campinas, SP, Brazil.

1.2.2 Papers in International Journals

- de Araujo, J. P., Arantes, L., Duarte, E. P., Rodrigues, L. A., and Sens, P. (2019). Vcube-ps: A causal broadcast topic-based publish/subscribe system. *Journal of Parallel and Distributed Computing*, 125:18–30.

1.3 Organization of the Manuscript

The rest of this thesis is organized as follows.

Chapter 2 presents some background knowledge on the different principles exploited throughout this thesis. It covers some important concepts in distributed systems such as causality of events, existing structures to track causality, causal order of messages, broadcast, and Pub/Sub systems. The virtual hypercube-like topology (*VCube*) and the algorithm used for building distributed spanning trees over it, which are used by both contributions of the thesis, are also presented in this chapter.

Chapter 3 summarizes some related work on the construction of spanning trees over hypercubes, causal broadcast, message bundling in dissemination protocols, and Publish/Subscribe systems.

Chapters 4 and 5 cover the two contributions of the thesis. The first one presents the causal aggregation protocol, while the second one presents *VCube-PS*. Both of them include the respective algorithms and their descriptions as well as evaluation performance results from experiments conducted on top of *PeerSim*.

Finally, ***Chapter 6*** concludes this thesis and proposes some future research directions.

Chapter 2

Background

Contents

2.1 Introduction

This chapter presents some concepts and existing approaches aiming at a better understanding of the thesis. It begins by discussing the impossibility of keeping track of time in a distributed system where a global clock is not available. In distributed applications, processes cooperate among themselves to perform tasks, often requiring to communicate with each other as a single group. Therefore, a communication service which offers a message broadcast primitive that enables a node to send a message to all others ensuring some order of message delivery is extremely important for those applications. It is worth remarking to remark that in this chapter and throughout the remaining of the thesis, the words nodes and process are interchangeable.

Thereby, initially, Section 2.2 introduces the notion of causality, through which it is possible to establish a relation of cause and effect between events, and classical approaches used to timestamp logical time. Section 2.3 summarizes the different types of broadcast related to message ordering. Both contributions of this thesis involve tree-based broadcast which respect the causal order of broadcast messages. These trees are built on top of a virtual hypercube-like topology called *VCube*. Section 2.4 presents the main characteristics of *VCube* and also shows how to build distributed spanning trees on top of *VCube*.

Lastly, Section 2.5 presents the principle of Publish/Subscribe, which is the subject of the second contribution of the thesis.

2.2 Causality in Distributed Systems

The idea of time is one of the basis of our way of thinking. It helps us to organize our daily activities by assigning to them duration and an execution order. The notion of temporal order is particularly useful in computer systems as well. In daily life, humans keep track of the physical time using loosely synchronized clocks (e.g. wrist watches) while centralized computer systems have physcical clocks. On the other hand, when it comes to distributed systems, it is impossible, due to clock drift, to ensure that machines' physical clocks are always perfectly synchronized.

A distributed system can be modeled as a set of N processes $(p_0, p_1, ..., p_{N-1})$ which communicate by message-passing and where each process performs a sequence of events. Events are considered to be atomic and they can be classified in three types: *send*, *receive*, and *internal* events. The latter affect only the process where they occur, and events at a same process are totally ordered by program order. Differently, *send* and *receive* events result in information exchanged between processes (Raynal and Singhal, 1996).

Let e and e' be two events. The relation between events in distributed systems was introduced by Lamport (1978), resulting in the well-known "happened-before" relation, denoted by $\rightarrow$, the smallest transitive relation on the set of events of a system which satisfies the following conditions:

- if e and e' occur in the same process and e comes before e', then $e \rightarrow e'$;
- if e is the *send* event of a message m and e' is the *receive* event of m, then $e \rightarrow e'$;
- if $e \rightarrow e'$ and $\exists\ e'' : e' \rightarrow e''$, then $e \rightarrow e''$ (transitive relation).

The "happened-before" relation can also be seen as a relation of cause and effect between events, for instance, if $e \rightarrow e'$, we also say that event e causally precedes event e'. On the other hand, if neither $e \rightarrow e'$ nor $e' \rightarrow e$ hold, then neither of them causally affects the other (Schwarz and Mattern, 1994). In this case, considering no global time, it is impossible to say which event occurs before the other. The events are said to be *concurrent* ($e' \parallel e''$). Note that the above relations define only a partial ordering of events, because for concurrent ones it is impossible to state which one happened first.

Figure 2.1 shows the causality ("happened-before") relation between events in a system with three processes. Global time increases from left to right, each dot represent an event $e_{i,j}$ (the

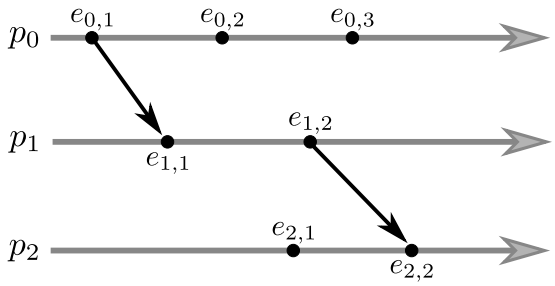


Figure 2.1: Relation between events in a distributed computation with three processes.

j^{th} event that took place in process p_i), and the transmission of a message is given by an arrow connecting a *send* event at a process with its corresponding *receive* event at another process. In the figure, event $e_{0,1}$ from process p_0 causally precedes $e_{0,2}$ and $e_{0,3}$, since $e_{0,1}$ occurred earlier in the same process. It is also straightforward to see that $e_{0,1}$ causally affects $e_{1,1}$, since they represent the sending and the corresponding reception of a message. Moreover, because $e_{1,1}$ causally affects $e_{1,2}$, $e_{0,1}$ also causally affects $e_{1,2}$ (transitive property). As a general observation, an event e causally precedes another event e' if, in the figure, there exists a left-to-right path starting in e and ending at e' (Mattern, 1989). Thus, even if in the figure event $e_{0,2}$ takes place earlier than $e_{2,2}$ regarding the global time, they are considered concurrent ($e_{0,2} \parallel e_{2,2}$).

2.2.1 Logical Clocks

Several authors have then proposed the use of logical time to detect causal precedence relation between events (Mattern, 1989; Fidge, 1988; Schwarz and Mattern, 1994). From an abstract point of view, a logical clock is just a way of timestamping. The concept of logical clocks was initially proposed by Lamport (1978) with the goal of partially ordering events in distributed systems.

Formally, a system of logical clocks consists of a time domain T and a logical clock C. The elements of T are partially ordered over a relation $<$. The logical clock C is a monotonic function that maps an event e of the set of events E to a timestamp $C(e)$ of T, such that the following condition holds:

Let $e, e' \in E$ be two distinct events of a distributed computation, timestamped by $C(e)$ and $C(e')$ respectively, then:

$$e \rightarrow e' \Rightarrow C(e) < C(e')$$

When the system of logical clock C satisfies the above condition, it is said to be *consistent with causality* (Schwarz and Mattern, 1994). On the other hand, it is said to *characterize causality*, if the following condition holds:

$$e \rightarrow e' \Leftrightarrow C(e) < C(e')$$

2.2.1.1 Scalar Clocks

Basically, scalar logical clock, proposed by Lamport (1978), associates an event with a scalar value. Every process p_i keeps a scalar variable C_i which represents p_i's logical clock. The following rules must be respected by scalar clocks:

R1 - Before executing an *internal* or *send* event, process p_i increments its local clock:

- $C_i \leftarrow C_i + d \qquad (d > 0$, although generally $d = 1)$

R2 - When p_i executes a *send* event, the sent message is piggybacked with the current value of C_i.

R3 - Upon execution of a *receive* event where a message with timestamp C_j is received, process p_i does the following:

1. $C_i \leftarrow max(C_i, C_j)$;
2. Execute **R1**.

Scalar clocks do not detect concurrence, thus two events may seem to be ordered even if they are in fact concurrent. Hence, if $C(e) < C(e')$, it is impossible to say whether the events are causally related or not. In order words, *Lamport* clocks are *consistent with causality*, but they do not *characterize* it.

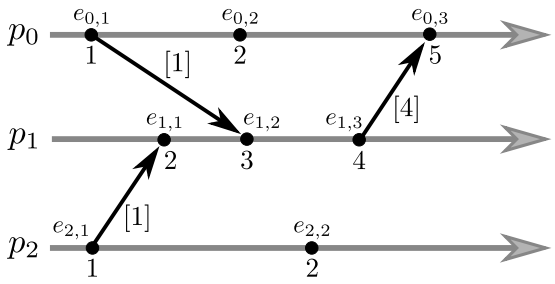


Figure 2.2: Example of a scalar (Lamport) clock for three processes.

Figure 2.2 depicts the evolution of scalar clocks in a distributed computation with three processes. To this end, each message from p_a to p_b also contains the value C_a. In the figure, events $e_{0,2}$ and $e_{1,3}$ are an example in which scalar clocks fail to characterize causality. Even if $C(e_{0,2}) < C(e_{1,3})$, these two events are in fact concurrent.

2.2.1.2 Vector clocks

The concept of *vector clocks* was independently proposed by Mattern (1989) and Fidge (1988). Differently from *Lamport* (scalar) clocks which use a single scalar variable to logical time, the logical clock of a process p_i consists of a vector V_i of size N, where N is the number of processes in the system. Entry $V_i[i]$ corresponds to the local clock of p_i, while for $j \neq i$, $V_i[j]$ is the knowledge p_i has of the local time of p_j.

All entries V_i are initially reset and they are updated according to the following rules:

R1 - Before executing an *internal* or *send* event, p_i increments its own local clock:

1. $V_i[i] \leftarrow V_i[i] + d \qquad (d > 0$, although generally $d = 1)$

R2 - When p_i executes a *send* event, the sent message is piggybacked with a copy of V_i.

R3 - Upon executing a *receive* event corresponding to the reception of a message from p_j, process p_i updates its own local vector clock V_i with the vector V_j received with the message:

1. $\forall k \in [0, N-1] : V_i[k] \leftarrow max(V_i[k], V_j[k])$;
2. Execute **R1**.

In order to compare two vectors, the following relations are defined: Let V_e and $V_{e'}$ be the vector clocks associated with the events e and e' respectively. The following relations hold (Schwarz and Mattern, 1994):

- $V_e \leq V_{e'} \Leftrightarrow \forall k \in [0, N-1] : V_e[k] \leq V_{e'}[k]$
- $V_e < V_{e'} \Leftrightarrow V_e \leq V_{e'}$ **and** $\exists k \in [0, N-1] : V_e[k] < V_{e'}[k]$
- $V_e \parallel V_{e'} \Leftrightarrow \neg(V_e < V_{e'})$ **and** $\neg(V_{e'} < V_e)$

Schwarz and Mattern (1994) proved that vector clocks characterize causality of events with which they are associated. Thus, for two events e and e' with vector clocks V_e and $V_{e'}$ respectively:

- $e \rightarrow e' \Leftrightarrow V_e < V_{e'}$
- $e \parallel e' \Leftrightarrow V_e \parallel V_{e'}$

Figure 2.3 shows the same time diagram of Figure 2.2. Contrarily to scalar clocks, by comparing the value of the vector clocks of p_0 and p_1, it is possible to know that $e_{0,2} \parallel e_{1,3}$, i.e., they are concurrent.

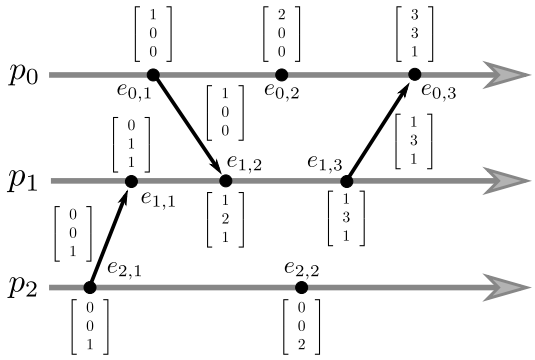


Figure 2.3: Example of vector clocks used to keep logical time of three processes.

Charron-Bost (1991) proved that the causality of events of a distributed system with N processes can only be *characterized* by using a vector of at least N entries. Therefore, vector clocks are not scalable.

2.2.2 Causal Order of Messages

When processes exchange messages by means of *send* and *receive* events, the communication network may assume one of the three existing models: *FIFO* (First-In First-Out), *Non-FIFO*, or *Causal Ordering* (CO) (Kshemkalyani and Singhal, 2008). In the FIFO model, the channel between any two processes p_i and p_j acts as a FIFO message queue and, in this case, message ordering is preserved by the channel. Differently, in the Non-FIFO model, channels do not guarantee that reception order of messages will be the same as their respective sending order.

The causal ordering model comes from Lamport's "happened-before" relation (Section 2.2). Hence, in order to support this model, it is necessary to satisfy the following property:

- Assuming $send(m)$ and $receive(m)$ as the send and receive events of a message m respectively, if $send(m') \rightarrow send(m'')$ and m and m' have the same destination, then $receive(m') \rightarrow receive(m'')$ (Raynal et al., 1991).

This property guarantees that the reception order of causally related messages arriving at the same destination process is consistent with the causality relation between the messages. Both contributions presented in this thesis apply causal order.

According to this definition, causal order is also transitive. Thus, a message can directly or indirectly precede another. For two messages m and m' sent to the same destination, message m directly (immediately) precedes message m' (denoted $m \prec_{im} m'$) if (1) the send event of m causally precedes the send event of m' and (2) there exists no message m'' such that the send event of m causally precedes the send event of m'', and the send event of m'' causally precedes the send event of m' (Prakash et al., 1996). On the other hand, an indirect dependency is obtained applying the transitive property.

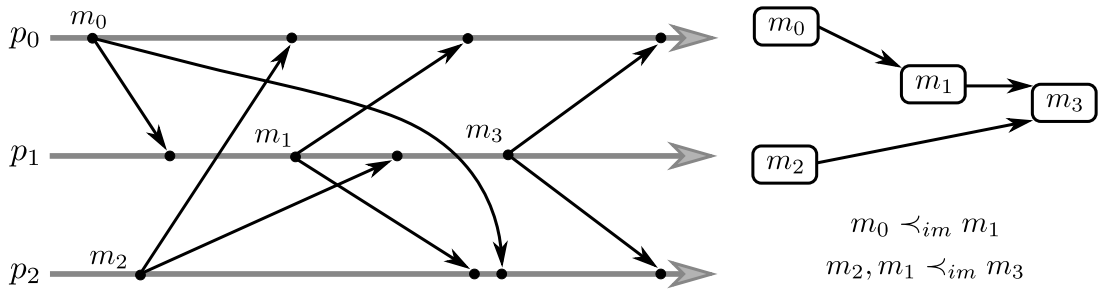


Figure 2.4: Example of transmission of messages and how they are related with respect to the causal order.

Figure 2.4 shows the possible relations between messages with respect to the causal order. On the left side of the figure, there is a timing diagram for a system with three processes (p_0, p_1, and p_2) with the sending and reception of some messages and, on the right, the graph with message dependencies. Initially, it is possible to observe that messages m_0 and m_2 are concurrent ($m_0 \parallel m_2$) because neither $m_0 \rightarrow m_2$ nor $m_2 \rightarrow m_0$ hold. Due to the transitive property, there is an indirect dependency between m_0 and m_3 ($m_0 \rightarrow m_1 \rightarrow m_3$), while there is a direct (immediate) dependency between m_0 and m_1 ($m_0 \prec_{im} m_1$) and m_3 is directly preceded by both m_2 and m_1 ($m_2, m_1 \prec_{im} m_3$). Lastly, the figure presents an example of causal order violation: although $m_0 \rightarrow m_1$, at process p_2, m_1 is received before m_0.

2.2.3 Causal Barrier

Causal barriers (Prakash et al., 1996) do not present the constraint of having a structure proportional in size to the number of processes as vector clocks do. Compared to vector clocks, causal barriers are a weaker way of representing causality, since the latter keeps less information about the causal history of a message. The advantage of the *causal barrier* approach is that it

does not control causality based on nodes' identity but by using direct dependencies of messages which also renders the algorithm more suitable for dynamic environments. Only if a message is directly preceded by messages received from every other process in the system (worst case), causal barrier and vector clock have the same size.

The *causal barrier* of m (cb_m) consists of the set of messages that directly precedes m. In the example of Figure 2.4, $cb_{m_1} = \{m_0\}$ and $cb_{m_3} = \{m_2, m_1\}$. Note that since m_0 precedes m_1 that precedes m_3, m_0 is an indirect dependency of m_3, not included, therefore, in cb_{m_3}. Therefore, compared to vector clocks, causal barriers lose information about causality. By using vectors, it is possible to know that $m_0 \rightarrow m_3$, but causal barriers do not keep this information because m_0 is not a direct dependency of m_3.

2.3 Broadcast

Many distributed applications, such as Publish/Subscribe systems, parallel applications, client-replicated servers, etc., require group communication support (service) where a process, by calling a single primitive, can send a message to all (***broadcast***) or to many (***multicast***) processes of the system.

Although some systems provide the broadcast and multicast primitives at network layer, they can be emulated by applying multiple one-to-one message transmissions in upper layers. However, in both cases, there are issues concerning the order in which messages are received and the reliability of the primitive in presence of failures.

As both contributions of this thesis are based on broadcast, in particular the causal one, in the following, some basic concepts are summarized.

2.3.1 Broadcast Basic Specifications

Basically, a broadcast communication support should offer to the application two primitives:

- BROADCAST(m): allows a process to send a message m to all processes of the system including itself. It is implemented by sending m to all processes by using point-to-point communication primitives.
- DELIVERY(m) is the event at which a message m is given to the application by process p_i.

After the *reception* of a message, its *delivery* to the application may be delayed in order to satisfy some condition (e.g., ordering) (Birman et al., 1991). Thus, the *receive* event of a message m ($receive(m)$), defined in Section 2.2.2, represents the arrival of message m from p_j at p_i through the channel between p_i and p_j and DELIVERY(m) is executed once all necessary conditions to render the message to the application are satisfied. It is worth highlighting that the delivery of message m by p_i is causally preceded by m's reception at p_i, i.e., $receive(m)$ at $p_i \rightarrow$ DELIVERY(m) by p_i.

These primitives are non-blocking: upon calling BROADCAST(m), p_i is not blocked waiting for all the processes to receive m, while DELIVERY(m) is called by p_i only upon notification that it received m ($receive(m)$) and that all necessary conditions for delivery were satisfied.

A process is said to be *correct* or *fault-free* if it has not crashed during the whole execution, otherwise it is *faulty*. Considering that channels are reliable and there is no failure, a broadcast service must ensure the following properties:

- ***Validity***: every broadcast message is eventually delivered by all processes.
- ***Integrity***: no message is delivered to a process more than once (no duplication), and only if it has been previously broadcast by some process (no creation).

2.3.2 Message Ordering

In a distributed system, a set of broadcast messages may reach each destination in a different order due to, for instance, latency variations or processing of participating processes. Hence, it is important to provide mechanisms that ensure that messages will respect a consistent delivery order with regard to the broadcast order of these messages. Basically, there exist three broadcast ordering of messages in the literature (Figure 2.5):

- ***FIFO Order***: it requires that messages broadcast by the *same source process* to be delivered in the order they were broadcast.

  Formally, if a process broadcasts m_1 before m_2, then no process in the system delivers m_1 after m_2.

- ***Total Order***: it requires messages to be delivered in the same order by all destination processes, no matter the sender (Défago et al., 2004).

  Formally, for any messages m_1 and m_2, if a process p_i delivers m_1 before m_2, then no process p_j delivers m_2 before m_1.

- ***Causal Order***: all processes must deliver messages by respecting the causal order of broadcast messages (see Section 2.2.2). Note that, causal delivery order implies FIFO order.

  Formally, if a process broadcasts message m_2 after it has delivered another message m_1, then no other process in the system can deliver m_1 after m_2.

2.3.3 Reliability

In a distributed system prone to node failures or message loss, a broadcast service as defined previously cannot guarantee that all processes will deliver all the broadcast messages.

As mentioned in Section 2.3.1, a basic broadcast service must satisfy the properties of validity and integrity. A *reliable broadcast* should satisfy the following three properties:

- ***Validity***: if a correct process broadcasts m, then it eventually delivers m.
- ***Integrity***: no message is delivered to a process more than once (no duplication), and only if it has been previously broadcast by some process (no creation).

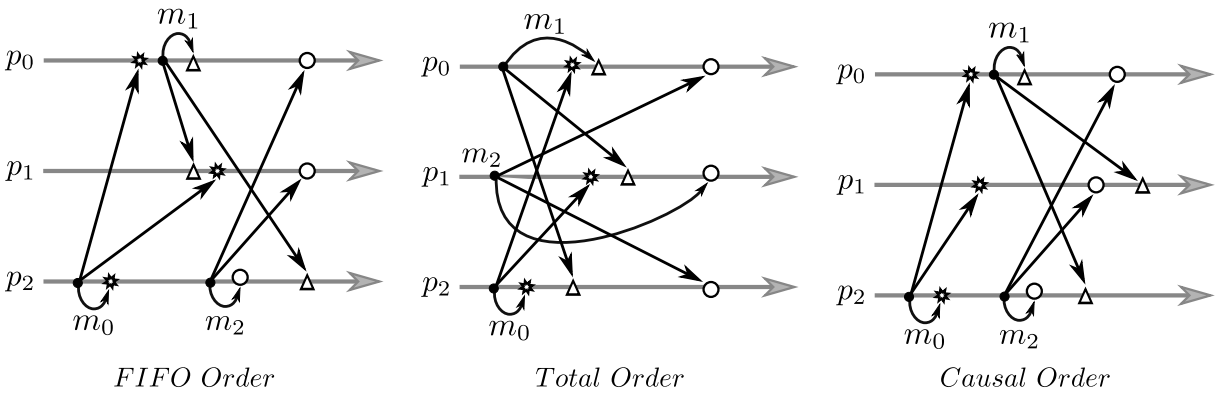


Figure 2.5: Examples of broadcast of messages with respect to the three ordering.

- ***Agreement***: if a correct process delivers m, then m is eventually delivered by all correct processes.

A reliable broadcast sets no condition on messages delivered by faulty processes, contrarily to *uniform reliable broadcast* which modifies the agreement property in order to state the expected behavior in presence of faulty processes:

- ***Uniform Agreement***: if a process delivers m, then m is eventually delivered by all correct processes.

Different broadcast properties may be combined in order to comply with an application specification. For instance, an *atomic broadcast* (ABCAST) is a reliable broadcast which guarantees the total order of delivered messages (Birman et al., 1991).

The system models of the two contributions of this thesis consider that processes do not fail and channels are reliable. Thus, the proposed solutions do not include reliability.

2.4 VCube

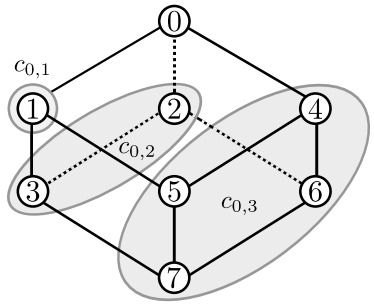
VCube (Duarte et al., 2014) is a distributed diagnosis algorithm. In a system consisting of N processes, unique identified from 0 to $N-1$, the identity of all processes is globally known, and the network is fully connected (complete graph). *VCube* organizes the correct processes of the system in a virtual hypercube-like topology, presenting, thus, logarithmic properties for the distance between nodes and node degree. A process i (also called p_i) groups the other $N-1$ processes in $d = \log_2 N$ clusters forming a *d-VCube*, such that each cluster s ($s = 1, .., d$) has size 2^{s-1}. The ordered list of processes in each cluster s is denoted by $c_{i,s}$ as follows, in which $\oplus$ denotes the bitwise exclusive *or* operator (xor).

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

A process i tests another process in the $c_{i,s}$ to check whether it is correct or faulty. It executes a test procedure and waits for a reply. If a reply is received within an expected time interval,

the monitored process is considered to be alive. Otherwise, it is considered to be faulty. If later it detects its mistake, it corrects it.

Virtual hypercube for node 0



The $c_{i,s}$ table for 8 nodes

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

Figure 2.6: *VCube* hierarchical organization.

Figure 2.6 shows the hierarchical cluster-based logical organization of $N = 8$ processes connected by a 3-*VCube* topology as well as a table which contains the composition of all $c_{i,s}$ of the 3-*VCube*.

Let's consider process p_0 and that there are no failures. The clusters of p_0 are shown in the same figure and, in this case, they are organized as a perfect hypercube. Each cluster $c_{0,1}$, $c_{0,2}$, and $c_{0,3}$ is tested once, i.e., p_0 only performs tests on nodes $1, 2, 4$ which will then inform p_0 about the state of the other nodes of the respective cluster.

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 j is the first fault-free process in $c_{i,s}$. Thus, any process (faulty or fault-free) is tested at most once per round, and the latency, i.e., the number of rounds required for all fault-free processes to identify that a process has become faulty is $\log_2 N$ in average and $\log_2^2 N$ rounds in the worst case.

2.4.1 Spanning Trees Over *VCube*

Besides providing a diagnosis algorithm, the logical hypercube organization of *VCube* can be exploited to dynamically buid spanning trees over which messages are broadcast. *VCube's* cluster hierarchy enables the construction and use of distributed spanning trees which comprise all correct nodes. More importantly, for the same set of correct nodes, trees can be organized differently, depending on the process that is chosen to be the root of the tree. It is also important to note that when the tree is built using all nodes of a complete hypercube, the tree is a binomial one (Vuillemin, 1978).

In order to send a message over *VCube*, the procedure is similar to the one used for failure detection. It uses first fault-free processes and sender's information to choose the next hop of a message. The resulting tree keeps the logarithmic properties of *VCube*, having maximum height equals to $\log_2 N$, and up to $\log_2 N$ children per process.

Let's consider $d = \log_2 N$ the dimension of *VCube* which is also the height ($h = d$) of the related spanning tree. For broadcasting a message m, node i sends m to the first correct node

of each of its clusters $c_{i,s}, \forall s \leq h$, to which i is linked. Upon receiving m, each of these nodes j becomes the root of a sub-tree whose height is $h = s - 1$. Therefore, if j is not a leaf ($h \neq 0$), it applies the same sending procedure of i's and so on. For instance, based on the *VCube* of Figure 2.6, the spanning tree over which m_0 will travel due to its broadcast by node 0 is shown in Figure 2.8(a).

Auxiliary functions: by exploiting the cluster organization of *VCube*'s virtual hypercube topology, it is possible to express some inference rules regarding the relation between nodes. Thereby, in order to easily build spanning trees, the following functions are offered to each node i:

CLUSTER(j, k): returns the index s of the cluster of node j that contains node k, ($1 \leq s \leq \log_2 N$). For instance, in Figure 2.6, CLUSTER$(0, 1) = 1$, CLUSTER$(0, 2) =$ CLUSTER$(0, 3) = 2$, and CLUSTER$(0, 4) =$ CLUSTER$(0, 5) =$ CLUSTER$(0, 6) =$ CLUSTER$(0, 7) = 3$.

FIRSTCHILD(j, s): returns the first correct node in $c_{j,s}$ (Figure 2.6), i.e., the first node of c_j, s which is linked to j. For example, FIRSTCHILD$(0, 1) = 1$, FIRSTCHILD$(1, 2) = 3$, and FIRSTCHILD$(1, 3) =$ FIRSTCHILD$(7, 2) =$ FIRSTCHILD$(4, 1) = 5$. On the other hand, if for instance, node 5 fails FIRSTCHILD$(1, 3) =$ FIRSTCHILD$(7, 2) = 4$, and FIRSTCHILD$(4, 1) = \perp$ (*null*).

CHILDREN(r, h): used by the broadcast protocol to either (1) obtain the children of node i in the spanning tree rooted at node r whose height is h or (2) to build spanning trees. The function returns the first correct child of each cluster $c_{i,s}$ of i, $\forall s \leq h'$, where h' is the height of the sub-tree of i in the tree of r (see Algorithm 1).

Algorithm 1 Children of i in r's tree

```
1: function CHILDREN(node r, height h)
2:     if i = r then
3:         return FIRSTCHILD(i, s) | 1 < s ≤ h
4:     else
5:         return CHILDREN(FIRSTCHILD(r, CLUSTER(r, i)), CLUSTER(r, i) − 1)
```

When i is equal to r, CHILDREN(r, h) simply returns its h children. Otherwise, the function recursively searches node i in the tree of r using the cluster of r where i is present. When the sub-tree rooted in i ($i = r$) is found, its respective children are returned. For example, if node 4 wants to know its children in the tree rooted in node 2, it invokes CHILDREN$(2, 3)$ which will recursively call CHILDREN$(6, 2) \rightarrow$ CHILDREN$(4, 1) = \{5\}$ (see Figure 2.7).

CHILDREN(r, h) function is also used for the construction of spanning trees. In order to broadcast message m, node i becomes the root of the spanning tree and sends m to its $\log_2 N$ children (CHILDREN$(i, \log_2 N)$). Upon the reception of m, j, a child of i, becomes the root of a sub-tree of i's tree with height CLUSTER$(i, j) - 1$. Note that the number of children of a node also decreases by one in relation to its parent's cluster. Hence, every node $k \in$ CHILDREN$(j,$ CLUSTER$(j, i) - 1)$, i.e., every child of j in relation to a tree where $j's$ parent is i, receives m from j and this proce-

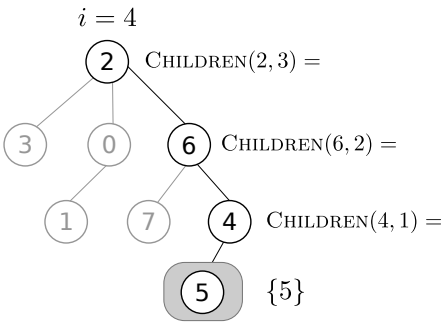


Figure 2.7: Function CHILDREN used to find the children of node 4 in the tree rooted at node 2 (fault-free system).

dure continues until m is received by all nodes that do not have children (leaves of the spanning tree).

For instance, in order to broadcast message m_0 in a fault-free system (Figure 2.8(a)), node 0 calls CHILDREN$(0, \log_2 N) = \{1, 2, 4\}$ and sends m_0 to them. Upon receiving m_0, node 1 does not forward m_0 since CHILDREN$(1, 0) = \emptyset$, node 2 forwards it to CHILDREN$(2, 1) = \{3\}$, and node 4 to CHILDREN$(4, 2) = \{5, 6\}$; Node 5 does not forward m_0 since CHILDREN$(5, 0) = \emptyset$. Node 6 forwards it to CHILDREN$(6, 1) = \{7\}$ while node 7 does not forward it since CHILDREN$(7, 0) = \emptyset$. Considering a second example in which nodes 2, 4, 6 are faulty, node 0 forwards m_0 to CHILDREN$(0, \log_2 N) = \{1, 3, 5\}$. Node 3 is a leaf node (CHILDREN$(3, 1) = \emptyset$), and node 5 sends the m_0 to CHILDREN$(5, 2) = \{7\}$. The resulting spanning tree is shown in Figure 2.8(b). Lastly, in order to depict the differences in the organization of trees rooted at different nodes, the spanning tree (without failures) rooted at node 2 is shown in Figure 2.8(c).

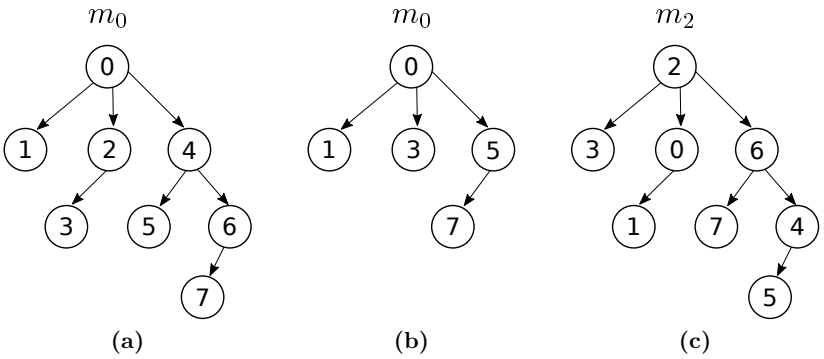


Figure 2.8: Function CHILDREN used to build a complete tree (without failures) rooted on node 0 (a) and with nodes 2, 4, and 6 faulty (b). The complete spanning tree for node 2 is presented in (c).

The approach of building spanning trees rooted at the broadcast source node presents better scalability when compared to single rooted approaches like (Kim et al., 2010; Wang et al., 2012), that organize the nodes of the system in a single static distributed spanning tree. The latter has the drawback that the root can become a bottleneck since all message broadcasts start from it. Figure 2.9, where all nodes of the system broadcast messages at a given rate, confirms the scalability of both approaches. In the case of a single tree, for a number of nodes greater than

128, the root of the tree starts queuing messages because it cannot process all of them as fast as the input broadcast request rate. On the other hand, if each node has its own broadcast tree, the load of messages is better distributed among the nodes. Furthermore, reception latency scales well for an increasing number of nodes/messages.

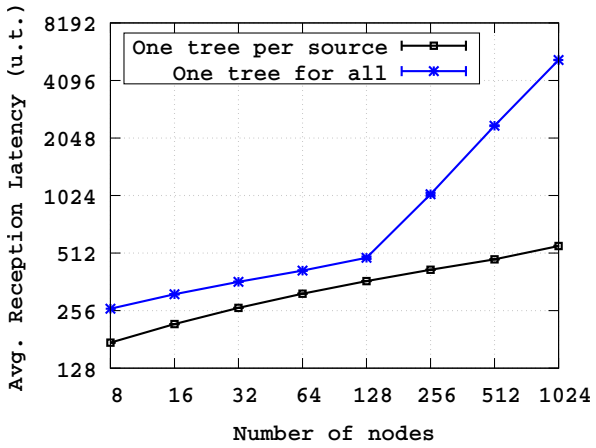


Figure 2.9: Reception latency for systems with different number of nodes using two different tree approaches: one single tree or multiple trees (one per source node) (de Araujo et al., 2017).

2.5 Publish/Subscribe Systems

Asynchronous dissemination of information is a key feature in many recent distributed applications. The Publish/Subscribe (Pub/Sub) paradigm has emerged as a suitable middleware solution for this challenge due to its decoupling properties and scalability (Astley et al., 2004; Esposito et al., 2013). A Pub/Sub system consists of distributed nodes in which one or more *publishers* produce messages that are consumed by *subscribers*. It is necessary to point out that in the literature, messages may also be denoted "events".

Communication between publishers and subscribers is conducted on an overlay infrastructure, which is generally composed by a set of nodes that organize themselves for ensuring the delivery of published messages to all (preferably only) interested subscribers. Therefore, publishers and subscribers exchange information asynchronously, without interacting directly (Baldoni et al., 2005; Esposito et al., 2013). They might even not know each other. Publishers do not get blocked while publishing messages, and similarly, subscribers are asynchronously notified about an incoming message, while performing some other concurrent activity.

According to Eugster et al. (2003), the decoupled production and consumption of events is a desirable characteristics because it removes explicit dependencies between different parts of the systems. Moreover, it reduces the necessary coordination and the resulting system is well suited for distributed environments since they are asynchronous by nature.

In order to receive messages, subscribers must inform the system about its interests. There are different ways of doing it, leading to different levels of expressiveness, as well as necessary processing to match messages. Two of the most widely used models are the *topic-based* (Castro

et al., 2002; Zhuang et al., 2001; Gascon-Samson et al., 2015; Zhao et al., 2013) and *content-based* (Cugola et al., 2001; Bianchi et al., 2010; Eugster et al., 2003) ones.

In the topic-based model, subscribers share a common knowledge on a set of available topics and every published message is labeled with one of these topics. A subscriber can register its interest in one or more topics, and thus she/he receives all published messages related to these topics. The concept of topics is close to the idea of "group communications" as exploited, for instance, by Birman et al. (1991). On the other hand, in the content-based model, messages are composed of multiple attributes, and subscribers express their interests by specifying constraints over the values of these attributes (Eugster et al., 2003).

The advantage of the topic-based model is that events/messages can be statically grouped into topics, the diffusion of messages to subscribers is usually based on multicast groups, and the interface offered to the user is simple. Even if it offers limited expressiveness for subscribers (Baldoni et al., 2005), the topic-based approach is widely used by applications such as chat message systems, Twitter, mobile devices notification frameworks (e.g. Google Cloud Messaging), and many others.

A topic-based system basically offers an interface consisting of three function associated to a topic t: SUBSCRIBE(t), UNSUBSCRIBE(t), and PUBLISH(t, m). While the first ones are used by a node to register and unregister its interest in messages related to t respectively, the latter is invoked by a node to publish a new message to the subscribers of t. The Pub/Sub system presented in Chapter 5 of this thesis is a topic-based one.

2.5.1 Message Dissemination and Delivery

A simple, but not always effective, way to disseminate messages to subscribers is *flooding*, through which messages are sent to the entire system, no matter the node (Baldoni et al., 2005). This approach needs a minimal amount of routing information, however it does not scale in terms of message overhead (number of messages transversing the network). Carzaniga et al. (2001) show that flooding is not a feasible strategy when the system presents a high churn rate because all membership modification needs to be informed to all nodes.

A more effective approach is to organize the nodes in a logical dissemination topology such as open cube, rings, tree, etc., and broadcast messages by exploiting such topology (e.g., HOMED (Choi et al., 2004)). Another solution is to build dissemination logical structures over a *structured overlay* infrastructure, where physical nodes are mapped into virtual keys obtained from a virtual key space, such as Pastry (Rowstron and Druschel, 2001) and Chord (Stoica et al., 2001) distributed hash tables (DHTs). These virtual keys are used to form a structured graph that is used to transmit messages and membership information. For instance, Scribe (Castro et al., 2002) is a topic-based Pub/Sub that builds dissemination trees over Pastry.

Examples of Pub/Sub systems for both approaches are introduced in the next chapters.

Some Pub/Sub systems (Cugola and Picco, 2006; Zhao et al., 2013) build an overlay with nodes denoted *brokers*, which behave like servers, and are responsible for disseminating messages and keeping information about subscriptions. Publishers and subscribers are connected to brokers. Publishers send messages to brokers which are responsible for forwarding these messages

to the interested subscribers. Furthermore, in order to optimize the organization of the system and reduce inter-broker communication, each broker may store just a subset of the subscriptions trying, for instance, to group subscriptions of a given topic in the same broker or organize subscriptions geographically. However, the topology formed by the *brokers* is generally assumed to be managed by an administrator, limiting its application to scenarios where topology changes are assumed to be rare (Baldoni et al., 2005).

Logical topologies and structured overlays tolerate system dynamics better than broker approach. In the former, a path between any two virtual nodes is supposed to exist despite the possibility of continuous arrival/departure of nodes (node churn).

The Pub/Sub system proposed in Chapter 5 of this thesis, the *VCube-PS*, exploits the hypercube logical organization of nodes provided by *VCube*. For broadcasting published messages, spanning trees composed only by subscribers and rooted at the publisher node are dynamically built. Due to dynamics of nodes, the spanning trees may temporarily have relay nodes.

Messages must be delivered only by interested subscribers. However, it might happen that non-subscriber nodes receive published messages, depending on the dissemination structure. In this case, the Pub/Sub support must filter those messages, not delivering them to the application. In this thesis, these non-subscribers nodes are called *relays*. Although relay nodes do not deliver messages, they are responsible for providing paths between subscribers. For instance, in Scribe (Castro et al., 2002), a dissemination tree of a topic may be composed by relays and subscriber nodes.

2.6 Conclusion

This chapter presented some important existing concepts which are exploited by this thesis. Both contributions use distributed spanning trees built on top of the virtual topology provided by *VCube*. *Vector clocks* are used in Chapter 4 to implement a new *causal broadcast* protocol where causally related messages are combined within a single message with no overhead, when their dissemination paths intersect. Chapter 5 presents a new topic-based *Pub/Sub system* that dynamically builds per-publisher spanning trees and that uses *causal barriers* to implement causal order. Works related to both causal broadcast and Pub/Sub systems are discussed in Chapter 3.

Chapter 3

Related Work

Contents

3.1 Introduction

This chapter focuses on some existing works that are related to this thesis. Initially, works related to both contributions of this thesis are discussed: Section 3.2 contains some works that

build distributed spanning trees on top of hypercube topologies, followed by different solutions for causal broadcast, organized according to the type of structure they use to enforce causal order (Section 3.3).

Different forms of reducing communication overhead by bundling messages are presented in Section 3.4, since this is the principle of the contribution of Chapter 4. Lastly, Section 3.5 presents some existing publish/subscribe systems, classified in terms of dissemination structure or message ordering feature.

3.2 Spanning Trees Over Hypercubes

In order to provide a broadcast mechanism, different strategies may be employed, such as organizing nodes in logical overlay (Rowstron and Druschel, 2001; Stoica et al., 2001) or using gossiping (Kim and Ahn, 2006), for instance. In this thesis, nodes are logically organized in a hypercube-like overlay and, on top of it, distributed spanning trees are built to broadcast messages.

In (Chang, 1982), the authors propose a tree structure for asynchronous communication on top of a general graph. Any node that initiates the broadcast of a message is the root of its broadcast tree and reply messages are sent from leaves towards the root to confirm the reception of the message. The author presents a set of rules for transversing the graph without creating loops and also a synchronization mechanism through which a node waits for reply messages for every message it has forwarded before sending its own reply.

Hélary and Mostefaoui (1993) exploit *open cubes* which are basically hypercubes where some edges are removed in order to obtain a binomial tree (Vuillemin, 1978). In case of failure, positions of nodes in the binomial tree are reorganized keeping logarithmic properties. Such a structure is used to propose a token-based fault tolerant algorithm where the root of the tree is the node which keeps the token and, by reorganizing the tree, the less a node requests to enter the critical section, the further it is from the root of the tree. Differently, in this thesis, every time a message is broadcast, a tree rooted on the broadcasting node is dynamically created whose organization is different for each root node. It is worth remarking that spanning trees built over *VCube* (see Section 2.4) have the structure of *open cubes* (binomial trees).

In (Wu, 1996), nodes are organized in a hypercube and a fault-tolerant binomial spanning tree is built over it. A tree with faulty nodes can be recursively rebuilt but, in this case, a special control message is used to notify the nodes about the tree reconstruction. During this process of reconstruction, other messages are not treated. In the Pub/Sub system presented in Chapter 5 of this thesis, although the system is considered to be fault-free, changes in topics membership require recontruction of the tree.

One of the major issues in large scale multicast applications is the amount of control information. For instance, ACK messages might lead to the *ACK implosion problem* (Crowcroft and Paliwoda, 1988), where the system is flooded by acknowledgement messages. Aiming at avoiding such a problem, Liebeherr and Beam (1999) present *HyperCast*, a protocol which uses a logical hypercube topology to disseminate control messages of multicast groups. Note that

HyperCast is not used for data transmission, even if this could be easily implemented. By using a hypercube topology, every node needs to keep only a partial membership knowledge (up to $\lceil \log m \rceil$ neighbors, where m is the number of nodes in the membership group). Messages are disseminated through binomial spanning trees built on top of the hypercube, such that the root of the tree is the sender of the control message. Every node notifies its neighbors about its presence by periodically sending ping messages. However, in case of failures or joins/leaves, the procedure used to keep the mapping between physical and logical address can temporarily lead the hypercube to an unstable state, where more than one node is associated to a same logical address.

Rodrigues et al. (2014) propose a reliable broadcast service for an overlay based on a hypercube-like topology. In this approach, an underlying service monitors the nodes and, in case of failure, the topology self-reorganizes itself. Spanning trees are dynamically built on top of the overlay, such that, analogous to the approach presented in this thesis, the source of each message is the root of its own broadcast tree. If failures happen, spanning trees may change during a broadcast. However, the last received message from each source must be locally stored by each node, because in the case of failures, retransmissions may be necessary in order to ensure that all correct nodes will receive a given message. Moreover, every node except the root sends and `ACK` message to its parent in the tree to confirm that all its children have received a message, which increases message overhead.

Properties of multiple rooted spanning trees built over hypercube-like topologies are also discussed by Yang et al. (2015), where the authors highlight its applicability in fault-tolerant broadcast. They exploit enhanced hypercubes (Tzeng and Wei, 1991), which are hypercubes with complementary edges used to improve distance between nodes and provide extra links in case of failure. On top of this topology, the authors build multiple spanning trees rooted at a given node r, such that for any other node v, the paths from r to v in different trees have no common intermediary node.

3.3 Causal Broadcast

This section discusses some causal broadcast protocols existing in the literature. Different solutions have been proposed in order to ensure causal order of broadcast messages and many of them aim at reducing the amount of control information sent in messages.

3.3.1 Message History

The first causal broadcast protocols of the literature use a trivial strategy to ensure causal order: every message m broadcast by a node p also includes a ***message history*** comprising all messages received by p since its last broadcast (Birman and Joseph, 1987). Upon reception, since m contains its causal preceding messages, the receiver can deliver, in order, all deliverable messages that precede m and then m itself. This approach has the advantage of delivering messages as soon as they are received. However, each node must keep a buffer of received messages between two broadcasts and the size of messages can be very large, becoming unpractical.

3.3.2 Vector Clocks

Instead of piggybacking message histories, Schiper et al. (1989) present an implementation of causal broadcast based on ***vector clocks*** (Fidge, 1988; Mattern, 1989), which is later applied by Birman et al. (1991). In this approach, every process p_i has a local vector clock V_i and every message carries a copy of its sender's vector clock ($m.vc$). Before sending a message m, p_i increments $V_i[i]$ and attaches V_i to the message. Upon reception of m from process p_j, process p_i must delay the delivery of m until (1) it has delivered all messages from p_j that precede m, and (2) it has delivered all messages delivered by p_j before the latter sends m. Formally:

$$\forall k \begin{cases} (1)\ m.vc[k] = V_i[k] + 1, & \text{if } k = j \\ (2)\ m.vc[k] \leq V_i[k], & \text{otherwise} \end{cases}$$

When process p_i delivers the message m sent by p_j, it updates its vector clock: $V_i[j] = V_i[j]+1$. By applying vector clocks, all causal dependencies among broadcast messages can be easily tracked. However, the size of a vector clock depends on the number of processes in the system (or the number of nodes participating in the communication, in the case of multicast). Thus, vector clocks are not scalable and may take a large amount of space in messages.

3.3.3 Reducing Message Size

In order to reduce the size of vector clocks sent within broadcast messages, several works propose approaches to reduce the amount of causal information (Birman et al., 1991; Prakash et al., 1996; Baldoni et al., 1996; Cai et al., 2002; Evropeytsev et al., 2017). Besides proposing causal broadcast implementation using vector clocks, Birman et al. (1991) also show that, it is not always necessary to send the entire vector along with every message. They show that for a process p_i that broadcasts a message m, the latter needs to carry only the entries of vector clock V_i that were modified since the last broadcast by p_i. Each process p_i must have an extra data structure to keep track of the modified entries of V_i. In the case where only some processes broadcast the majority of the messages, this compression technique can reduce substantially the size of the transmitted vector timestamp. However, in the worst case, if all nodes frequently broadcast messages, few or no compression will be possible. Thus, the compression technique is not always advantageous, since it depends on the locality and frequency of broadcasts. The first contribution of this thesis applies such a compression strategy.

Causal barriers (Section 2.2.3) exploit the transitive property of vector clocks for reducing the amount of causality information transmitted within a message. The advantage of the causal barrier approach is that it does not control causality based on processes' identifiers (per process vector entry) but by using direct dependencies of messages. The first broadcast protocol that employs this type of structure was proposed by Prakash et al. (1996) and later optimized for multicast environments by Cai et al. (2002). Basically, they use local structures to track dependency information and include in the messages only direct dependencies. In the case of the solution presented by Prakash et al. (1996), each process locally applies a matrix to store information about the last messages delivered by other processes, while in (Cai et al., 2002),

vector clocks keep information about messages that the process has already delivered.

Figure 3.1 shows the difference in the amount of causal information carried by each message using Birman's compressed vector clock or causal barriers. In the experiment, messages are broadcast at a given rate and the source of each message is selected according to a uniform distribution. In the case of vector clocks, compared to causal barriers, several entries of the vector clock are likely to change between consecutive broadcasts of the same node, resulting in larger overhead (in average 47 entries per message, in a system with 1024 nodes). On the other hand, causal barriers keep only direct dependencies, which is, in average, less than 6 entries per message in a system with 1024 nodes. However, it is important to remark that, at the cost of a larger message size overhead, vector clocks carry more information about the causal history of a message because it takes into account indirect dependencies. For the same data presented in the figure, results also show that vector clocks carried up to 31 indirect dependencies for the same message (in the scenario with 1024 nodes).

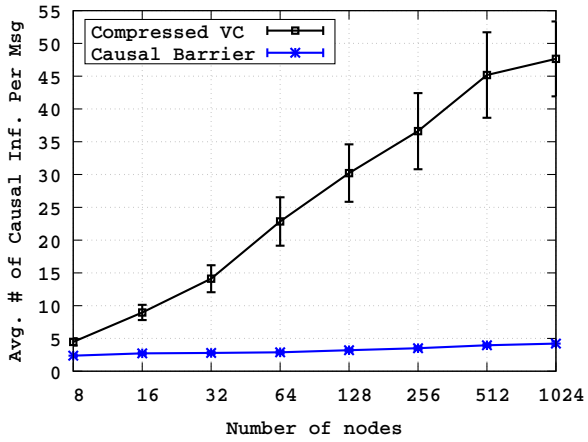


Figure 3.1: Average amount of the causal information carried by messages when using causal barriers and Birman's compressed vector clocks. Results obtained using data from (de Araujo et al., 2017, 2018).

Several types of applications (e.g. distributed multimedia services) need information to be delivered in causal order and within an expected time. Motivated by this observation, Baldoni et al. (1996) propose a causal broadcast with time constraints (Δ-causal ordering). In the protocol, any message received after an interval Δ from its creation is considered lost. A message is delivered only if all its direct causal dependencies are either (1) delivered or (2) lost (creation time + Δ > receive time). Although this work results in a Δ-causal ordering protocol with reduced communication overhead, it relies on a global clock to control delivery time constraints. Since the size of a message is one of the parameters that has an impact in latency for message delivery, the authors apply causal barriers in order to reduce the amount of causal information carried by a message.

The work presented by Evropeytsev et al. (2017) aims at reducing the amount of control information carried by messages in hierarchical networks where powerful nodes, denoted "super nodes", communicate among themselves while the others are connected to a super node. A group is a set of nodes connected to a same super node and nodes only communicate to each

other through super nodes. Based on such an architecture and assuming non-FIFO channels, the authors propose a protocol where the size of a causal barrier is limited to either the size g of a group (in the case of group communication) or to $n - g$ (inter-group communication), where N is the number of nodes in the system. Although this approach reduces the size of messages, the hierarchical architecture requires nodes with heterogeneous power.

3.3.4 FIFO Channels – Small/No Control Information

The work presented by Friedman and Manor (2004) guarantees causal broadcast by flooding messages through reliable FIFO channels of network overlays. The idea is that whenever a process p_i receives a message m for the first time, p_i forwards m to all its neighbors in the overlay excluding the process from which it received m. Since channels are FIFO, this dissemination strategy ensures that messages traveling over the same channel respect causal order. Thus, messages do not need to carry any control information and they can be delivered as soon as they are received. It is important to remark that the causal delivery order can be violated if new links are dynamically added in the system due to the arrival of new processes.

Nédelec et al. (2018a) extend the above work of Friedman and Manor (2004) in order to cope with dynamics. In their approach, messages only travel through safe links, i.e., the ones which messages are already known to have arrived in causal order. New links are assumed to be unsafe and become safe after a ping phase: when process p_i adds a new (unsafe) link e_{ij} to process p_j, p_i sends a ping message to p_j through a path composed by safe links. Upon reception, p_j replies to p_i through the unsafe link e_{ij}. As channels are FIFO, when p_i receives the reply from p_j, p_i knows that no message sent before the ping phase will be received by p_j. However, no assumption can be made about the messages sent after the ping, and if p_i sends a new message through e_{ij}, it might lead to a violation of causal order by p_j. In order to solve this issue, p_i buffers the messages it delivers during a ping phase and, after the reception of the reply from p_j, p_i sends all buffered messages to p_j through the link e_{ij}. After this procedure, e_{ij} is safe, i.e., no message from p_i to p_j will arrive out of causal order. Compared to (Friedman and Manor, 2004), this approach presents the extra cost of using one buffer for each unsafe link but, on the other hand, it tolerates dynamics such as mobility of nodes.

In approaches that use vector clocks, all processes need to keep information about all other processes all the time. Nédelec et al. (2018b) observes that, since the amount of information stored per process increases linearly with the number of processes, this kind of solution becomes unpractical when dealing with very large systems. Therefore, they propose an approach based on reliable FIFO channels where any process knows the number of copies of a given message that it can receive. A message m is considered active for a process p_i between the first and last reception of m at p_i. As soon as m becomes inactive, p_i removes all control information related to m. When m is received for the first time at p_i, the latter delivers it and then keeps a temporary register which informs from each input neighbors it will still receive another copy of m. This solution works for static networks, but in order to cope with dynamics, the protocol also uses a set of control messages exchanged between processes that establish new links. By receiving these control messages, processes are aware about when the links is safe to receive new

messages, avoiding therefore, duplicated message delivery. Compared to Nédelec et al. (2018a), this approach increases the number of control messages, but it reduces local information per process from $O(N)$ to $O(I \cdot A)$, where n is the number of processes, I is the number of input links, and A is the number of active messages.

Bravo et al. (2017) proposed Saturn, a service for implementing causal consistency in geo-replicated systems where metadata used to track causality and payload are decoupled. This approach only considers metadata dissemination, assuming that any other mechanism can be used to propagate the payload itself. Clients are associated to datacenters and a datacenter knows the causal past of its clients. For a given operation (e.g. a data update that must be replicated to other datacenters) there exists a unique label, which is a fixed-size structure that identifies the operation. Propagation of labels generated at a same datacenter is managed by a logically centralized component, which collects labels and transmits them in a serialized order that is consistent with causal order. Trees are used to propagate labels through FIFO channels, where a node that initiates a broadcast is the root of its own tree. However, the process of serializing labels can induce false dependencies due to concurrent operations. Thus, the authors propose an optimization to define serialization strategies, such that the order of the labels is compliant with causality and false dependencies do not affect delivery latency. In the Chapter 4 of this thesis, trees rooted at the source of messages are also used for broadcast. However, different from (Bravo et al., 2017), we use causality information sent with the payload to calculate, at each hop, if it is possible to bundle causally related messages in order to reduce message size and network congestion.

3.3.5 Probabilistic Approaches

Motivated by the observation that, for some scenarios, a system can deliver most of messages in the causal order without any explicit control, Mostéfaoui and Weiss (2017) propose a probabilistic approach using reliable broadcast where, at the cost of a small rate of violations in the causal delivery order, it is possible to reduce the size of vector clocks. To this end, the authors extended the concept of *Plausible Clocks* (Torres-Rojas and Ahamad, 1999), a logical time structure of constant size. These clocks basically consist in associating several processes to the same entry of a vector clock and, although they do not characterize causality, they are scalable due to their constant size. On the other hand, in the work of Mostéfaoui and Weiss (2017), each process can also be associated to several entries of the vector clock.

Probabilistic causal broadcast protocol is suitable for applications where the missing of some causal relation information does not lead to lack of correctness, although there exists a trade-off between the size of the vector clock and the rate of causal order violations. Moreover, the size of the system not necessarily needs to be known, making it more suitable for dynamic environments. On the other hand, a probabilistic causal broadcast is not suitable for the implementation of the causal aggregation broadcast protocol presented in Chapter 4 of this thesis because the proposed aggregation mechanism aims at combining as much as possible causally related messages into a single message, thus requiring precise knowledge about the chain of causal dependencies of a received message, and not incomplete or partial ones.

3.3.6 Application-defined Causality

Bailis et al. (2012) discuss some scalability issues associated to traditional mechanisms used to track causal dependencies in terms of number of dependencies and the time necessary to check them. Differently from the traditional concept of causality, where the entire history of preceding messages may affect a new one, they propose the use of explicit or application-defined causality: a sub-set of the causal history which reflects only the causality in application level. For instance, in a public discussion group, even if a message causally depends on much older ones, in fact it depends only on those associated to its subject (from the point of view of application).

Blessing et al. (2017) exploit the idea of application-defined causality and propose a causal broadcast protocol where messages carry no causal information. They organize the actors (processes) of an application into a tree topology that guarantees (explicit) causal order delivery. Thus, the path used by the "causing" message must somehow be included in the path of the "caused" ones. However, FIFO channels are necessary and the organization of the tree is time-costly and application-dependent.

Table 3.1: Characteristics of several causal broadcast solutions.

Article	*C.O. Structure*	*Channels*	*Dynamics*	*Msg. Overhead*	*Local Mem.*
Birman et al. (1991)	Vector Clock	Reliable	✓	$O(N)$	$O(N)$
Prakash et al. (1996)	Causal Barrier	Reliable	✓	$O(N)$	$O(N^2)$
Cai et al. (2002)	Causal Barrier	Reliable	✓	$O(N)$	$O(N)$
Baldoni et al. (1996)	Causal Barrier	Unreliable	✗	$O(N)$	$O(N)$
Blessing et al. (2017)	Application-defined	FIFO	✗	-	-
Mostéfaoui and Weiss (2017)	Probabilistic	Reliable	✓	$O(R)$	$O(R)$
Friedman and Manor (2004)	-	Reliable FIFO	✗	$O(1)$	$O(N)$
Nédelec et al. (2018a)	-	Reliable FIFO	✓	$O(1)$	$O(N)$
Nédelec et al. (2018b)	-	Reliable FIFO	✓	$O(1)$	$O(I \cdot A)$
Bravo et al. (2017)	Labels	Reliable FIFO	✓	$O(1)$	$O(1)$

$R < N$, I is the number of input links, and A is the number of active messages.

Table 3.1 summarizes some of the causal broadcast approaches discussed in this section in terms of network characteristics, structure used to capture the causal relation between messages, and amount of information stored by the processes and transmitted within the messages.

3.4 Bundling Messages

This section discusses some approaches for optimizing communication by aggregating/bundling messages.

An important feature in broadcast algorithms is communication overhead. According to Chetlur

et al. (1998), the number of messages has a higher impact than the size of the messages in such an overhead. They show that the communication cost of one packet involves mainly (1) a component that varies with the size of the message ($w \cdot c$, where w is the size of the message and c is the cost for sending each unit of the message size) and (2) a static overhead cost s which is generally up to two times bigger than c. Following this metric, in order to send two messages of size w_1 and w_2 separately, the overall communication cost is $2s + c \cdot (w_1 + w_2)$. On the other hand, if the contents of the two messages are bundled into a single message of size $w_1 + w_2$, communication cost drops to $s + c \cdot (w_1 + w_2)$. Hence, it is more efficient to communicate two or more data items using a single messages. However, the majority of existing approaches delays the sending of a message using timers in order to wait for more messages which induces message delivery delays.

3.4.1 Parallel discrete event simulators

Chetlur et al. (1998) argue that there exists a trade-off between the gain associated to less frequent communication and the potential problems that message delivery can suffer when delaying messages in order to bundle them within a single message. They present a new approach that is applied to parallel discrete event simulators, which suffer from high overhead due to frequent communication. In this case, messages with the same destination that must be sent in close temporal proximity can be bundled/aggregated and, in order to cope with delay issues, they compare two different policies to adapt the buffering time for bundling between a same source and receiver. The first policy simply considers a fixed time window for bundling messages with destination to a same source and, whenever it expires and there exist buffered messages, the latter are bundled and then sent. A second one dynamically adapts the time window according to the rate of arriving messages, better supporting burst communication, for instance.

Another work which aims at reducing communicating overhead at parallel discrete event simulators is presented by Wang et al. (2013). They show that on cluster of multi-core nodes, communication latency between nodes is much higher than intra-node communication, no matter the size of the message. They propose that events from different local threads of one node should be grouped before transmission to another node. Differently from (Chetlur et al., 1998), which bundles messages with respect to one specific destination, this approach creates one thread that is responsible for bundling messages from one multi-core node addressed to another multi-core node.

3.4.2 Reduction of Energy Consumption in Wireless Sensor Networks

Aggregation strategies can also be used to maximize lifetime of energy-limited networks, such as Wireless Sensor Networks (WSNs), in which intense communication leads to rapid energy depletion (Akkaya et al., 2008). In such networks, generally a tree rooted at a sink node (or base station) is used to gather information from leaf nodes. However, in case of sink failure, the entire network becomes unavailable. In order to make the network more reliable, Yestemirova and Saginbekov (2018) present an approach for using aggregation in WSNs with multiple sinks,

where the same set of aggregated information is sent to several sink nodes. They organize the network in the form of a spanning tree, in which the root node also takes part of a backbone that connects all sinks. Data is aggregated from leaves towards the root of the tree, with respect to known sensing intervals and the root node is responsible for sending all aggregated data to the sinks through the backbone. As a result, the number of transmissions is close to the number of original messages.

3.4.3 Application Layer Bundling

According to Sianati et al. (2015), even if powerful devices are used, large scale cloud environments present undesirable end-to-end delivery time. Since the cloud is deployed over an underlying distributed infrastructure, one possibility would be optimize the communication service of such infrastructure. However, the authors argue that such an optimization is not possible in public clouds. In this case, they propose a message bundling approach applied to the application layer, such that no modification is necessary in the underlying network. In this protocol, instead of sending messages directly to the underlying network, application processes send messages to special bundler processes, which buffer messages according to their destination and use timers. When the timer of a bundler process expires, it bundles the buffered messagse within a single one and sends it to their respective destination.

3.4.4 Bundling Over Peer-to-Peer Overlays

Structured peer-to-peer overlay (P2P) networks such as Distributed Hash Tables (DHTs) can also take advantage of message aggregation/bundling in order to enhance performance. Saroiu et al. (2002) show that in a P2P network the average session time of a node is quite short, which leads to a huge amount of membership modification events being transmitted through the overlay. Gupta et al. (2004) present an aggregation-based solution to such a problem. In their approach, the circular logical identifier space of the P2P system is divided into slices, each of them 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 then aggregates them into a single message before sending them to the other slice leaders. Similarly, a leader can aggregate messages it received from other leaders before routing them to the other nodes of its slice.

Hidalgo et al. (2010) propose an aggregation protocol for P2P for scenarios where nodes may dispatch several look-up operations at a short period of time. One example of this scenario is given by routers constantly connected to the overlay and portable devices that connect to a router in order to access the overlay. In this case, the router is responsible for handling all requests of its connected devices. The authors modify Pastry (Rowstron and Druschel, 2001) in order to support aggregation, such that a message M is actually composed of k look-up messages. Messages are aggregated taking into account their logical proximity in the logical ring, such that, in the ordered set represented by M, the first message is the one whose destination is the closest one to the aggregating node. Like Gupta et al. (2004), they also propose a multi-slice mechanism but, in this case, the logical space defined by the interval between the key of the first and the last

message of M is divided in S slices. Thereby, message M itself is split in S messages M_s and all messages M_s are dispatched in parallel. However there exists a trade-off between the number of slices and the impact of aggregation: more slices reduce the transmission delay and overlay hops, since messages are sent to addresses closer to their destinations, but increases network traffic because fewer messages will be aggregated.

Another bundling approach designed for structured overlays is presented by Shudo (2017). The author states that it is possible to reduce forwarding cost by bundling messages whose paths overlap. A node that wants to dispatch a bundle of messages chooses as the next hop the node that is part of the common path of all messages in the bundle. If such a common node does not exist, the message is split and re-bundled into smaller ones, according to their common paths. The process of splitting a larger bundle according to their paths is repeated until messages reach the destination nodes, resulting in a dissemination tree in which the root is the node that initiates the transmission. The closer a node is to the root, the higher the bundles it performs. Unlike this work, in the protocol presented in Chapter 4 of this thesis, messages are initially sent individually and, during propagation, are bundled based on causal order.

3.4.5 Timer-based Bundling Over *VCube*

Rodrigues et al. (2018) propose a tree-based best-effort broadcast protocol using dynamically built spanning trees rooted at the source of each message, on top of *VCube* (see Section 2.4) where messages that must be sent to the same destination are grouped into a single message. Although messages broadcast by different source will have trees organized differently, there may exist path overlaps. Messages that share a common path at some moment during their dissemination are bundled within a single message. The protocol uses timers to wait for such messages. When a node i receives a message m, a copy of m is stored at a buffer b_j for each children j of i in m's spanning tree. The timer t_j associated to each buffer b_j starts when b_j stores a first message. Buffer b_j is sent sent to j when (1) no more messages can be bundled due to the maximum limit size or (2) t_j expires. In the order to implement the best-effort protocol, every broadcast message also induces an ascending wave of acknowledgement messages (ACKs). These ACK messages can also be bundled following the same path overlap approach. Despite its performance improvement compared to a version of the protocol without message bundling, this solution still requires timers that should be fine-tuned according to the application load in order to present good performance. The causal aggregation protocol proposed in Chapter 4 of this thesis uses the same tree structure on top of *VCube*, but bundles messages without the use of timers. A message m stays in buffer b_j just the time necessary to receive all m's causal preceding messages with which m can be aggregated and sent to node j. Hence, a message is delayed only the time it would anyway need to wait before being delivered at the destination j if no aggregation approach was used.

Table 3.2 summarizes some characteristics of the bundling approaches presented in this section, in terms of organization of the network, nodes that perform bundling/aggregation of messages, and policies used for bundling. Note that all works use timers, differently from the new approach presented in Chapter 4.

Table 3.2: Characteristics of presented bundling approaches.

Article	*Organization*	*Timer*	*Aggregator*	*Policy*
Chetlur et al. (1998)	Cluster of nodes	✓	Every node	From node to node with fixed or dynamic timers
Wang et al. (2013)	Cluster of multi-core nodes	✓	Leader of a core	From threads of a core to another core
Yestemirova and Saginbekov (2018)	Tree over WSN	✓	Every node	Bottom-up tree aggregation with copies of bundled message to multiple sink nodes
Sianati et al. (2015)	Application layer aggregation	✓	Bundler processes	One bundle per destination process
Gupta et al. (2004)	Multi-slice P2P	✓	One bundler node per slice	Bundle membership change notifications from one slice to another
Hidalgo et al. (2010)	Multi-slice P2P	✓	Every node	Bundle look-up messages from one node to a same destination slice
Shudo (2017)	Tree over Structure Overlay	✓	Every node	Top-down split of bundled messages according to path intersections
Rodrigues et al. (2018)	Tree over Hypercube Topology	✓	Every node	Nodes bundle received messages before forwarding according to path intersections

3.5 Publish/Subscribe Systems

This section initially discusses, in Section 3.5.1.1, some topic-based Publish/Subscribe (Pub/Sub) systems which use trees, since *VCube-PS*, the system proposed in Chapter 5 uses trees to disseminate messages. Next, Sections 3.5.1.2 and 3.5.1.3 present some existing topic-based systems in the literature that rely on clustering subscriptions or gossiping algorithms to enhance delivery of messages, respectively. Trees are also used in content-based approaches, as shown in Section 3.5.2. Lastly, knowing that delivery order of messages is an important feature for many types of applications, Section 3.5.3 discusses Pub/Sub works that implement different ordering solutions.

3.5.1 Topic-based Publish/Subscribe

Even if the process of filtering messages is simpler in topic-based systems, reducing the existence of unnecessary hops and consequently delivery latency is the goal of different works, which exploit different topologies or dissemination algorithms.

3.5.1.1 Tree-based Approaches Over Peer-to-Peer Overlays

Many tree-based Pub/Sub system have been implemented on top of logical overlays, being ***Bayeux*** (Zhuang et al., 2001) one of the first proposed solutions. It is built on top of Tapestry (Zhao et al., 2001), a structured overlay which implements a Distributed Hash Table (DHT), and it is a *rendezvous*-based system, where each topic has a single multicast tree. In Bayeux, every node that joins a topic sends a message that is propagated all the way to the root of the topic's tree,

which keeps a list of all subscribed nodes. Unsubscriptions are similar and messages must be sent to the root which is responsible for their propagation. However, the root node is a scalability bottleneck because it must store a potentially huge amount of subscription information, it is responsible for all publications of its topic, and it is also a single point of failure. In order to cope with these issues, Bayeux splits the root into several replicas that receive a disjoint partition of the membership set, selected according to logical locality.

Scribe (Castro et al., 2002) is a well-known decentralized topic-based Pub/Sub that constructs multicast trees on top of Pastry (Rowstron and Druschel, 2001) DHT. Among other characteristics, it uses less local information and creates less control information traffic when compared to Bayeux. Every node in Scribe is mapped to an address (n_{id}) from Pastry's circular key-space. Similarly, each topic also has its own identifier g_{id}, generally obtained by hashing the name of the topic. A node whose n_{id} is numerically the closest to a group's g_{id} becomes the *rendezvous point* for the group, which is the single root of the multicast tree of the group.

The operations provided by Scribe work as it follows: in order to subscribe to a topic t, a node s must join t's multicast tree. In this case, node s uses Pastry's routing function to forward a subscription message towards t's rendezvous point. During the routing process of a subscription message, at each hop through the overlay, there exist two possibilities: (1) if the current node h is already part of t's tree, it adds s as its child table and the forwarding process finishes or (2) node h will register node s as its child in t's tree and then h will forward a new join message to the next hop towards the *rendezvous* point. In this case, although node h is not actually interested in t's messages, it becomes a *relay* member of t's tree because it is a node whose logical address is in the path between s and the root of the tree. Similarly, unsubscriptions are propagated towards the root, because once a node u unsubscribes from a topic, other nodes that were relays of u may also be removed from the tree. However, if node u has some children in the topic, it will continue to participate in the topic's multicast tree as a relay node. Whenever a node publishes a new message to a topic t, the former forwards the message to t's *rendezvous* node and then, starting the *rendezvous* node, every node that receives the message will forward it to the nodes in its child table that are also part of t's tree. It is important to note that during the multicast process there may exist false positives induced by the relay nodes that take part of the tree only to connect a subscriber to the rest of the tree.

In terms of reliability, in Scribe, children can detect a faulty parent due to missing heartbeat messages and then rejoin the the tree using the same subscription process discussed above. Scribe also copes with failures in *rendezvous* points by replicating them to nodes logically close. However, it is worth remarking that Scribe specifies no particular message delivery order. Differently from Scribe, the Pub/Sub presented in Chapter 5 of this thesis ensures the causal order for messages published to the same topic and, instead of a single root per topic, it dynamically builds trees rooted on the source of every message. Moreover, in the absence of node churn, there is no relay node, i.e., all nodes of a tree are subscribed to the tree's topic. Otherwise, in presence of subscription dynamics, there might exist some temporary relays.

Multicast tree built on top of peer-to-peer networks must be continuously updated because peers join and leave the overlay. Therefore, there exists a maintenance cost that increases both

the time necessary to deliver messages to subscribers and the network traffic. Motivated by this observation, Li et al. (2011) proposed a system based on Scribe called ***DRScribe***, which aims at reducing costs by optimizing the routing procedure and then minimizing the number of relay nodes. Contrarily to Scribe, in which the next hop of a subscription message is the neighbor with the closest address to the one of the rendezvous point, DRScribe takes into account the subscriptions of neighbors. However, since propagating all subscriptions to all neighbors is costly in terms of traffic and space, DRScribe exploits a probabilistic data structure, called Bloom filter (Bloom, 1970), which is a fixed-size bit vector that uses a set of hash functions to map elements into the vector. Thereby, every node keeps a Bloom filter representing its own subscriptions and a set of filters for its neighbors in the overlay. Filters are updated when new neighbors' subscriptions are issued, but since there is no way to remove elements from such filters, they must periodically be re-initiated.

Girdzijauskas et al. (2010) observe that most of the existing DHTs implement node mapping following a uniform hashing process in order to provide load balancing and keep connectivity. However, as discussed previously for the case of Scribe, this uniform distribution of nodes in the logical space may induce the presence of relay nodes in multicast trees. Therefore, Girdzijauskas et al. (2010) propose ***Magnet***, a Pub/Sub system built on top of a small-world DHT named Oscar (Girdzijauskas et al., 2007). In this system, nodes with similar interests have logical addresses close to each other and, in order to implement such an interest-aware mapping, it uses a distributed membership service that periodically samples interests of some subscribers. When a node first joins Magnet or when it joins/leaves a topic, it invokes the membership service informing its interests in order to receive its new position in the logical space. Message dissemination and tree maintenance are similar to Scribe, with the difference that the number of relays is reduced.

Motivated by the subscription management overhead that occurs when a topic's tree is updated due to subscription dynamics, Zhao et al. (2013) propose ***DYNATOPS***, a tree-based Pub/Sub whose trees are composed of brokers instead of nodes. Trees are built on top of Chord (Stoica et al., 2001), in which, similarly to Scribe, each topic is assigned a *rendezvous* point. DYNATOPS tries to reduce communication among different brokers by assigning to the same set of brokers those users that share similar interests. In this way, a broker is said to subscribe to a topic's tree if at least one of its members is subscribed to the topic. Similarly, it can be removed from the tree, when none of its members is subscriber. However, similarly to Scribe, there may exist relay brokers that take part of a tree just to provide a logical path between other brokers. Thus, in order to reduce or eliminate them, logical positions of brokers can be changed. However, in order to decide when and where to move brokers, the system relies on a logically centralized control that periodically monitors the system and, if necessary, assigns new logical addresses to the brokers.

Another approach that also reinforces the locality of nodes with similar interests in topic-based Pub/Sub is called ***Rappel*** (Patel et al., 2009). This solution uses two decentralized structures built on top of a peer-to-peer overlay: one manages the relationships between nodes and the second one corresponds to dissemination trees. Nodes use a gossiping algorithm to

propagate Bloom Filters that encode their interests and, based on these filters and logical network proximity, they choose neighbors that share common interests. Each topic has its own single-rooted dissemination tree, which does not contain relay nodes. Differently, the new solution proposed in Chapter 5 uses the same dissemination structure to send both publications and control information and load balancing is improved by dynamically building differently organized trees rooted on the source of each message.

3.5.1.2 Clustering Solutions

TERA is a topic-based Pub/Sub proposed by Baldoni et al. (2007) built on top of a peer-to-peer system which clusters peers subscribed to the same topic. The system uses, at a lower layer, a global overlay network that connects all nodes. On top of it, several topic overlays (clusters) are built, such that each of them is a subset of the global overlay and connects all subscribers of the same topic. The goal of TERA is to propose an approach for the routing of messages to their target clusters, considering that non-subscribers can publish to a topic. Each cluster is assigned an access point, i.e., a node responsible for communicating with other cluster's access points. The latter is chosen according to a uniform probability and each node has a table where it stores the identifers of known access points. Whenever a message is published, if the access point of the topic is not yet known, the message is routed following a random walk through the global overlay until it finds the identifier of the access point of its topic and, once the message is received by the access point, it is disseminate to the other members of the cluster by flooding the topic overlay. However, because of the limited size of the table where nodes store the identifiers of known access points, it is possible that it will not store the identifier of every single topic. Thus, the table is updated according to the popularity of the topics: the access points of clusters whose topics receive frequently new subscriptions are less likely to leave nodes' tables. As a consequence, inactive topics will disappear. The authors also propose a solution for merging topic overlays that belong to the same topic. This situation can happen when more than one node subscribes to a topic which has no access point (or the topic does not exist).

Similarly, Chen et al. (2016) present ***OMen***, a DHT-based solution for clustering subscriptions in topic-based Pub/Subs deployed in datacenters with low churn rate. This approach also follows the principle that each topic must induce the creation of a connected sub-overlay comprising all subscribers of the topic. In this system, the authors tackle the problem of the maintenance of topic-connectivity in the overlay in the case of node churn (join, leave, or crash) and mainly in case of concurrent churns. OMen uses coordinator nodes that are responsible for proactively building backup sets that are used to restore topic overlay connectivity in case of churn. These sets are built using gossiping to gather information about the partial view of other nodes and must contain enough information about the network to be able to reestablish connectivity and, at the same type, ensure the existence of a sub-overlay per topic comprising only the subscribers of the topic.

BeaConvey (Chen et al., 2018) is topic-based system that uses small world topologies to devise a solution for keeping users with similar interests logically close. This system targets datacenter applications with infrequent churn and it relies on a centralized control which is

responsible for the maintenance of the overlay and keeping global knowledge of subscriptions. Nodes maintain only partial topology knowledge and the size of the path covered by any message is limited by $\log N$ (nodes in the overlay). Message dissemination is performed by dividing, at each hop, the address range a message must be delivered to, such that each sub-range is reached by a hop chosen by finding the sound balance between small routing overhead (no relay nodes) and reduced latency (exploit connectivity of the small world topology).

3.5.1.3 Other Topologies

Instead of using trees, ***PolderCast***, presented by Setty et al. (2012), uses an epidemic-based algorithm to create and maintain disseminating rings on top of a Peer-to-Peer Network. It is assumed a fully connected network in which nodes' identifiers are obtained from an ordered circular space. For every topic in the system there exists a ring (i.e., a sub-set of the network's circular space) connecting only the subscribers of the topic. Rings also contain some extra random links, in order to both accelerate propagation of messages in large rings and to cope with partitioning in case of node churn. Since every publisher of a topic must also be a subscriber of the topic, the entire forwarding of a message involves only interested nodes. Any node that receives a message for the first time (or creates it) forwards it to its neighbors and possibly some other random subscribers. New subscribers join a topic's ring by gossiping a subscription message through the network. Crashes and unsubscriptions are detected the same way by using ping messages. Unlike this approach, in *VCube-PS*, every node receives only one copy of each published message, which, compared to PolderCast, reduces message traffic.

Aiming at providing an intuitive subscription language, Cañas et al. (2015) propose a system called ***GraPS*** in which subscriptions are expressed using some connectivity criteria over a graph. For instance, considering an application where nodes represent partitions of a geographic area, the authors state that a classic topic-based system would use several subscriptions to cover an area, while GraPS could use a single one, with a distance metric in the graph to define area boundaries. The decentralized version of the solution is built on top of a fully connected network of brokers which filter and disseminate messages. An interest graph among the brokers is formed by exchanging information about subscriptions of the nodes connected to the brokers. Thus, when a message is published, it is sent to a broker, and the latter sends the message to its local interested nodes and to other interested brokers.

3.5.2 Tree-based Content Publish/Subscribe

HOMED is a content-based Pub/Sub proposed by Choi et al. (2004) that maps nodes to a logical hypercube and builds binomial trees to disseminate messages. In order to avoid unnecessary hops (relay nodes), nodes with similar interests should be neighbors in the hypercube. In this case, the interests of a subscriber are used to decide its position in the hypercube. The forwarding process of published messages is composed of two steps: first, the message is routed to any node whose interests matches the message and then, this node acts as the root of the binomial tree used to disseminate the message. Subscription is performed by choosing the position of the node and

informing its new neighbors while unsubscriptions are treated locally. Differently from *VCube-PS* (Chapter 5), delivery of messages does not respect causal order of published message broadcast.

The location or geometrical mapping of subscriptions can also be taken into account in the construction of dissemination structures for content-based systems. Bianchi et al. (2010) presented ***DR-Tree***, an approach where subscriptions and messages of a content-based Pub/Sub are represented geometrically and the tree itself acts as a spatial filter. The tree is based on R-Trees (Guttman, 1984), which are balanced trees where every non-leaf node keeps a minimum bounding rectangle (MBR) such that all children of a node fit in the node's MBR. By construction, this kind of tree does not present false negatives and false positives occur when a node's MBR is larger the its children's. DR-Tree is built on top of a peer-to-peer network, such that each node of the tree is under the responsibility of a peer and and each peer represents at least one subscription which is stored in leaf nodes. The root of the tree is the node with the largest MBR and internal nodes are created as MBRs for groups of subscriptions. Messages do not need to be sent to the root of the tree in order to be disseminated: a message created by a node n is sent downwards for the sub-trees whose MBR overlaps with the message's and upwards in order to find other nodes whose sub-trees are also interested in the message. The authors also present algorithms for the maintenance of the tree, in order to periodically recalculate MBRs and, if necessary, reorganize the tree. Later, in order to reduce delivery latency of DR-Tree, Arantes et al. (2010) add extra links to the tree to exploit interest proximity or enhance parallelism. Links are between brothers (nodes that share the same parent), to ancestors, and/or to the root node and, during dissemination, a node can forward messages directly to these extra linked nodes. In *VCube-PS* (Chapter 5), parallelism is achieved through the different organization of multiple trees, such that messages published by different sources will follow different paths.

Another approach called ***AP-Tree***, proposed by Wang et al. (2015), uses both textual and spatial information to organize subscriptions in a tree. This work targets subscriptions that consist of keywords and spatial information (e.g. cellphone discount within a 1 km radius from home) and subscription organization is adaptable according to their distribution: for subscriptions that are heavily overlapped in space, it is easier to filter them textually and, on the other hand, if they are scattered throughout the space, it is easier to primarily distinguish them by spatial coordinates. Moreover, AP-Tree also copes with moving subscriptions, i.e. if a user issues a subscription to some event close to the user's position, if the user moves, the subscription must be updated.

Table 3.3 summarizes some of the characteristics of the discussed Pub/Sub systems. None of them enforces causal delivery order for published messages. The table also shows the name of the simulator used for implementation or the structure used for deployment. Some systems that implement ordering of messages are discussed in the next section.

3.5.3 Message Ordering

Lumezanu et al. (2006) presented an ordering solution for topic-based Pub/Sub system that ensures the total order of messages sent to topics whose membership overlap. In the case of messages that are sent to unrelated topics, they may be delivered in any (potentially inconsistent)

Table 3.3: Characteristics of several publish subscribe systems.

System	***Type***	***Topology***	***Dissemination***	***Relay***	***Implement.***
Bayeux	Topic	DHT (Tapestry)	*RV* / Tree	✓	-
Scribe	Topic	DHT (Pastry)	*RV* / Tree	✓	-
DRScribe	Topic	DHT (Chord)	*RV* / Tree	Reduced	Sgaosim (Li and Gao, 2011)
Magnet	Topic	Small World DHT (Oscar)	*RV* / Tree	Reduced	-
DYNATOPS	Topic	Brokers over DHT (Chord)	*RV* / Tree	Reduced	Open Chord[1]
Rappel	Topic	Connected P2P	Tree (messages) Gossiping (control)	✗	PlanetLab[2] (deploy)
TERA	Topic	Global Overlay (GO) Topic Overlays (TOs)	Random Walk (GO) Flooding (TO)	✓	PeerSim (Montresor and Jelasity, 2009)
BeaConvey	Topic	Small World / Central Overlay Coordinator	Tree	✗	PeerSim
PolderCast	Topic	Fully Connected P2P	Epidemic / Rings	✗	PeerSim
GraPS	Topic	Brokers over Connected Overlay	Graph	✗	Padres (Fidler et al., 2010) (deploy)
HOMED	Content	Logical Hypercube	Tree	✓	-
DR-Tree	Content	Connected P2P	Distributed R-Tree	Reduced	-
AP-Tree	Content / Location	Centralized	Spatial-Textual Tree	✗	-

RV: *rendezvous* node. Reduced: the approach tries to reduce the number of relays.

order. The key feature of the approach is the presence of sequencers, which are logical nodes that assign sequence numbers to messages addressed to topics that share subscribers. New membership intersections between unrelated topics induce the creation of new sequencers and a group of sequencers forms a single loop-free path that connects all related sequencers. It is assumed FIFO channels between sequencers and the task of defining the sequence number of a message is distributed among the sequencers according to the last message sent to each intersecting topic. Thereafter, the message is sent to a dissemination tree and nodes use the sequence number of a message to decide when to deliver it. Furthermore, nodes of the same group see messages in the same order, which is consistent with the causal order provided that publishers are also members of the group.

By using Scribe (Castro et al., 2002) without inducing any message ordering, Baldoni et al. (2012) performed some experiments considering different topics, publishers, and subscribers. Results show that only 35% of the received messages follow the same sequence order. Instead of proposing a solution to guarantee a consistent delivery order to 100% of messages, they present a topic-basic Pub/Sub system where messages published on different topics are either delivered

[1] https://sourceforge.net/projects/open-chord/
[2] https://www.planet-lab.org/

in the same order to all subscribers or tagged as out-of-order (weak total order). The system uses reliable FIFO channels and the out-of-order detection algorithm uses logical timestamps. Although a simple solution for ordering messages would be the use of a special sequencer to timestamp all messages, this approach presents both a scalability bottleneck and a single point of failure. Therefore, they assumed that each topic is assigned a node that acts as its "topic manager", i.e., the sequencer of the topic. Moreover, topics identifiers should be totally ordered. The idea is that a group of sequencers comprising all sequencers of topics whose subscriber set intersect with the one of the published message will be responsible for creating the timestamp of the message, i.e., a table with an entry per sequencer. Upon reception, subscribers check if the message is in a coherent order according to its own local subscription clock (a structure that stores the timestamp of the last received message from each topic) and decides whether or not to tag the message as out-of-order. It is important to note that this approach could be extended to reorder messages by using buffers at reception.

A distributed total order protocol for a content-based Pub/Sub system is presented by Zhang et al. (2012). Differently from topic-based Pub/Sub systems, in content-based ones there is no explicit group where messages can be assumed as already ordered. The proposed solution consists of a detection phase, in which a broker determines if a message needs to be reordered and, if it is necessary, a group of brokers settles a consistent delivery order for the message. The simplest solution for a fault-free scenario is to use FIFO channels and, if publishers have a common broker in the path to all interested subscribers, total order is guaranteed. Otherwise, sequence numbers per publisher and information exchanged between brokers are used to decide if it is necessary to delay messages in order to ensure total order.

In (Malekpour et al., 2011), the variations of end-to-end delay of messages in a content-based Pub/Sub, directly related to out-of-order FIFO delivery, are measured. Values show that for most cases, FIFO order violations happen when sending interval of messages by the same sender is small and that end-to-end delay follows a hypoexponential distribution. Thus, based on such an analysis, the authors propose a probabilistic solution in which subscribers should decide to delay or not the delivery of a message in order to wait for some missing message previously sent by the same publisher. Nodes keep track of the messages sent per publisher using vector clocks. Thus, upon reception, if there is a gap between the sequence number of the message and the corresponding local vector clock entry, the receiver should wait for missing messages. However, this simple solution may lead to unnecessary delays, because some messages may not match the receiver's interests. In order to mitigate this problem, every message carries a history encoded in a Bloom filter which contains matching information about messages previously sent by the same publisher. Upon reception of message m from p, a subscriber s decides, using the filter if it will possibly receive another message sent by p before m and the waiting time is given by the hypoexponential distribution.

There exist some works like (Nakayama et al., 2016; Yamamoto and Hayashibara, 2017) that consider, as assumption, causal delivery of published messages, although neither of them provide solutions for specifically ensuring causal delivery order of messages of the same topic, as proposed by *VCube-PS* in Chapter 5 of this thesis. Nakayama et al. (2016) assume a topic-based system

where messages associated to the same topic are automatically delivered in causal order and that the same message can be sent to subscribers of several topics. In their proposal, messages from different topics are reordered at a subscriber s if they are causally related and if there exist intersecting elements between the topic set of each message and the subscriptions of s. For scalability sake, vector clocks are not included in messages, but only a similar "topic vector" that represents the number of messages sent per topic. The authors propose ordering rules based on this "topic vector" and using synchronized physical clock. The work presented by Yamamoto and Hayashibara (2017) merges partitions of a topic in a dynamic network, assuming that messages published in each partition are already causally ordered. This is the case for a scenario where an unstructured network is split due to, for instance, link problems and, therefore, subscribers of a given a topic can form disjoint partitions. Since nodes of different partitions continue to publish messages independently, the idea is that when a link between disjoint partitions is reestablished, messages published during the partitioning time will be exchanged. Then, in order to ensure causal order between partitions, each node must consider the vector clocks of messages of both partitions.

JEDI (Cugola et al., 2001) is an infrastructure for developing broker-based Pub/Sub solutions that ensure causal order. Published messages are acknowledged: an acknowledgement is sent by the receiver to the sender of a message in order to inform that the message was correctly delivered. After receiving such an acknowledgment, the sender can send the next message. Since publishers communicate with other clients through brokers, the latter inform publishers about the number of acknowledgments to wait for. Differently from this solution, which induces message traffic overhead, *VCube-PS* (Chapter 5) does not require these extra acknowledgments since direct causal dependencies of a message are included in the message itself using causal barriers.

Table 3.4 shows some of the characteristics of the ordering solutions discussed above. It is important to note that, among them, JEDI is the only one that ensures a delivery order that respects causality of published messages.

Table 3.4: Ordering characteristics of some publish subscribe systems.

Article	***Type***	***Topology***	***Channels***	***C.O. Structure***	***Delivery Order***
Lumezanu et al. (2006)	Topic	DHT	FIFO between sequencers	Sequencers	Total if topics overlap
Baldoni et al. (2012)	Topic	-	Reliable FIFO	Sequencers / Logical timestamps	Total or / tag out-of-order)
Zhang et al. (2012)	Content	Broker net.	FIFO	Broker communication	Total if interests overlap
Malekpour et al. (2011)	Content	-	-	Message history	Probabilistic FIFO
Nakayama et al. (2016)	Topic	-	-	Logical and physical times	Causal for overlapping groups
Yamamoto and Hayashibara (2017)	Topic	Unstructured	-	Vector clocks	Causal for merging partitions
JEDI (Cugola et al., 2001)	Content	Broker net.	FIFO	Confirmation messages	Causal

3.6 Conclusion

This chapter presented some existing works in the literature that are related to the contributions proposed by this thesis.

Initially, it was shown that hypercubes present logarithmic properties which are interesting for the construction and maintenance of spanning trees and that causal broadcast approaches have evolved trying to reduce the amount of information necessary to guarantee causal broadcast order.

Such causal broadcast protocols include probabilistic approaches, where a small rate of causal order violations can happen. Other protocols eliminate or reduce the inclusion of causal information in broadcast messages by exploiting characteristics of the underlying topology such as FIFO channels. By considering the point of view of the application, Bailis et al. (2012) redefine the concept of causal dependency, such that only a sub-set of the causal history is used.

Concerning the reduction of communication overhead in broadcast protocols, several solutions have proposed to bundle messages using timers that, at the cost of increasing latency, reduce message traffic.

Lastly, this chapter discusses several Publish/Subscribe systems that, among other characteristics, try to reduce or eliminate nodes that participate in the forwarding of a published message which are not interested in the message. Some Pub/Sub systems ensure some coherent message delivery order. However, even if a few works address causal order, as far as our knowledge, only JEDI (Cugola et al., 2001) actually guarantees causal delivery order to all published messages.

Unlike the bundling approaches presented in this chapter, the causal aggregation broadcast presented in Chapter 4 does not user timers. It exploits the causal relation between messages and common destination nodes to decide which messages can delayed for bundling. Furthermore, it delivers messages in causal order.

The new Pub/Sub system, *VCube-PS* (Chapter 5), ensures causal delivery order for messages published to the same topic. It also mitigates the impact of root bottleneck presented in spanning trees used by several topic-based systems presented in this chapter. While several systems use a unique tree per topic, *VCube-PS* dynamically builds trees rooted on the source node of each published message. Furthermore, in the absence of subscriptions dynamics, a tree built to publish a message contains only the subscribers of the message's topic. Otherwise, nodes that unsubscribe from a topic may behave for a while as relay nodes, but eventually they will not take part of the topic tree anymore.

Chapter 4

Causal Aggregation Broadcast

Contents

4.1 Introduction

As discussed in Section 3.4, the performance of communication protocols is particularly associated with the number of messages and not necessarily with their sizes. Several works try to optimize communication by delaying the forward of messages in order to bundle several of them before transmission. Although, even if this kind of approach reduces traffic, it increases end-to-end message delay.

Based on such a consideration, this chapter presents a new causal broadcast protocol which combines causally related messages into a single message, such that a message is delayed only when it is known that such a message can be bundled with causally preceding ones with no extra delivery delay. Moreover, the protocol guarantees causal delivery order.

The idea behind this protocol is to exploit how causal order violations happen and, at the same time, take advantage of such violations in order to bundle messages. In other words, the protocol exploits execution scenarios where indirect communication (messages relayed via intermediate nodes) is faster than direct communication. Known as *Triangle Inequality Violation* (TIV) (Adelstein and Singhal, 1995; Plesca et al., 2006), this kind of violation happens due to disparities in channels and network congestion. Existing studies on TIV show that this is a widespread and frequent problem (Lumezanu et al., 2009; Wang et al., 2007).

Figure 4.1 shows a case of TIV (channel delay $t_c > t_a + t_b$), such that node 2 receives messages out of the causal order and, therefore, some delay and additional treatment are imposed before delivering them to the application in the correct order. Let's also assume that node 2 is responsible for forwarding both messages m_0 and m_1 ($m_0 \rightarrow m_1$) to node 3 (not in the figure). Therefore, we can ask the question: What is the difference in delivery latency at node 3 if node 2 waits to receive both m_0 and m_1 and then send them into a single message compared to sending m_1 as soon as it is received and, later, m_0? There is no difference, since node 3 would have to wait for m_0 anyway to deliver both messages. However, by sending them together, node 2 reduces network traffic.

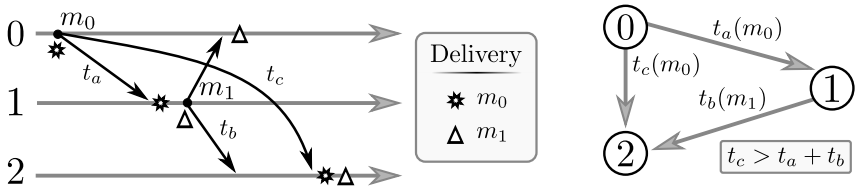


Figure 4.1: Example of TIV. Although $m_0 \rightarrow m_1$, m_0 is received first at node 2.

In order to delay the forwarding of m_1 to node 3, node 2 needs to know that node 3 will not receive m_0 through another path. In the proposed causal broadcast protocol, such a knowledge is acquired by exploiting inference rules to build spanning trees over *VCube* (see Section 2.4), used to disseminate messages to all nodes. When a node broadcasts a message, the protocol dynamically builds a *VCube* spanning tree rooted at that node. On the other hand, even if the organization of a tree depends on its root node, part of the paths of different trees intersect. Moreover, thanks to the above mentioned rules, every node knows how every other tree is constructed and it can, therefore, deduce intersections of different trees. Consequently, a node can delay the forwarding, to one or more of its tree's children, of those messages whose causal dependencies it knows that these children cannot satisfy them yet, since it is the responsible for forwarding these missing messages to them. Upon receiving the missing messages, the node aggregates all the messages and send them within a single message to those children.

Differently from some approaches discussed in Section 3.4, where messages are aggregated during a waiting time (implemented with timers), entailing extra delays to delivery latency, the proposed aggregation approach does not induce any overhead neither degrades performance as it is based on the principle that the sending of a message to a node is worthless if the latter will not be able to deliver it. Interestingly, due to such a reduction in the number of messages over the network, the average delivery latency can also be improved since there is less node contention.

Section 4.2 presents the system model adopted for the development of this protocol, followed by a detailed description of the protocol (Section 4.3) and its algorithm (Section 4.4). Comparison experiments with and without the proposed aggregation approach were conducted on top of *PeerSim* and Section 4.5 presents some evaluation results.

4.2 System Model and Definitions

The model used by the protocol presented in this chapter considers a distributed system composed of a finite set of $\Pi = \{0, .., N-1\}$ nodes with $N = 2^d$, where $d > 0$ is the dimension of *VCube* (see Section 2.4). Each node has an unique identifier (*id*) and may be called by its *id* or by p_{id}. Each single node executes a task (process) and a user of the system corresponds to a node. Therefore, the terms node, user, and process are interchangeable.

Nodes communicate by message passing through bidirectional channels. The topology of the connected (not necessarily fully) network must allow nodes to be logically organized as an hypercube interconnection network. Nodes do not fail and links are reliable. Thus, messages exchanged between any two processes are never lost, corrupted nor duplicated. The system is asynchronous, i.e., relative processor speeds and message transmission delays are unbounded.

The *source* of a message is the *id* of the node that broadcasts a message. As discussed in the basic specifications of broadcast protocols (Section 2.3.1), this model also distinguishes between the arrival of a message (*reception*) at a process and the event at which the message is given to the application (*delivery*). Note that only the latter respects the causal order of broadcast messages.

For sake of clarity, this chapter considers *message* the data message of the application/user to be broadcast and *packet* the message of the broadcast protocol. A *packet* can, thus, aggregate several *messages*. Moreover, throughout the text, the terms bundling, aggregating, and combining are interchangeable. They represent the act of creating a new packet which groups a set of messages. No modification is applied to the original messages.

4.3 Aggregating Causally Related Messages

The spanning trees used in this approach are built on top of *VCube*, as presented in Section 2.4.1. For every broadcast message, a distributed spanning tree rooted on the source of the message is created. Although for different root nodes, spanning trees are organized differently, their nodes may have some common children, i.e., some parts of the paths of two messages may intersect at a node. By exploiting this spanning trees intersection feature, a node can delay, to one or more of its children, the forwarding of the messages whose some causal dependencies it knows that the children in question cannot satisfy yet. In other words, node i will postpone sending m to a child node k if the following conditions are satisfied:

- $\exists$ message $m' : m' \rightarrow m$;
- Node i is responsible for forwarding both m and m' to node k;

- Node i knows that m will not be able to be delivered by node k because node i has not received/delivered m' yet.

Thus, according the above conditions, node i will send all the missing messages m' and m to node k aggregated into a single packet only after receiving the former. It is important to remark that even if the forwarding of m to k was delayed, such a postponement does not cause any extra delay in m's delivery by k.

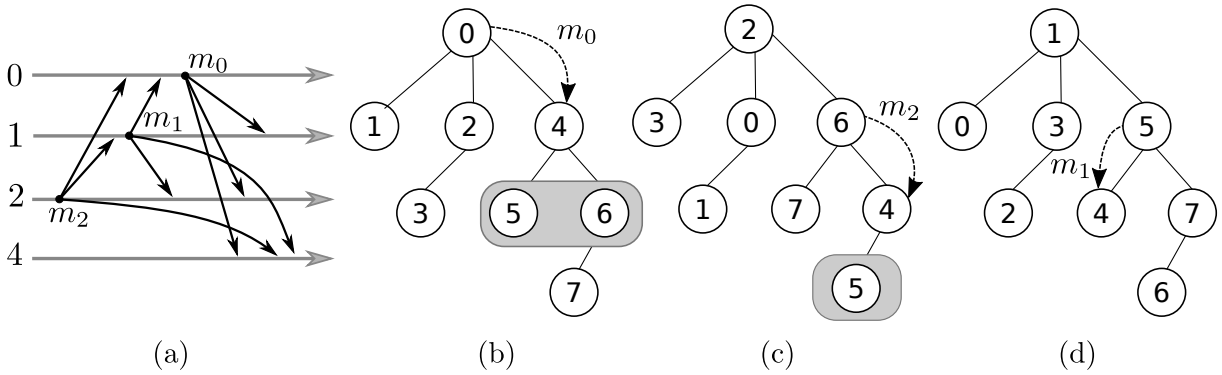


Figure 4.2: Example of spanning trees and intersection of tree paths

Let's consider Figure 4.2(a) where $m_2 \rightarrow m_1 \rightarrow m_0$. The broadcast of these messages by nodes 0, 2, and 1 dynamically builds different spanning trees, as shown in Figure 4.2(b,c,d), in a system with 8 nodes. In order to analyze how the protocol bundles messages, let's focus on node 4. Upon the reception of m_0, node 4 verifies that it has not received either m_2 or m_1 yet, as shown in Figure 4.2(a). Without the aggregation approach, node 4 would simply forward m_0 to its children in relation to the tree rooted in m_0's source (node 0), i.e., nodes 5 and 6 (see Figure 4.2(b)). However, as observed in Figure 4.2(b,c), node 5 is child of node 4 in both m_0's and m_2's trees and node 4 knows it because of *VCube*'s inference rules (by means of the function CHILDREN, defined in Section 2.4.1, page 17). Thereby, by applying the proposed aggregation approach, node 4 takes the following actions:

- m_0 is forwarded immediately to node 6 because the latter is not a common child of node 4 in the spanning trees of both m_0 and m_1;
- the forwarding of m_0 to 5 is delayed because m_2 precedes m_0 and node 4 has not received m_2 yet.

Upon reception of m_2, node 4 aggregates m_0 and m_2 within a single packet and sends it to node 5. Note that node 4 does not wait for m_1 before sending the packet to 5 because the latter is not a child of 4 in m_1's tree (see Figure 4.2(d)). In fact, due to the different organization of trees with different root nodes, node 5 is the parent of node 4 in m_1's tree.

When messages that could be aggregated due to intersecting paths are received in causal order, aggregation is unnecessary. For instance, if node 4 had received m_2 before m_0, the messages would not be aggregated to 5 since, in this case, upon reception of m_2, node 5 would be able to deliver it without depending on the reception of m_0.

Even if the forwarding of messages can be delayed for certain nodes and not to others, it is important to observe that, every message broadcast by a given process is received only once by each other process. No matter whether the message was bundled with others or not, it crosses only once each of the $N - 1$ links of the corresponding spanning tree.

4.4 Causal Aggregation Algorithm

In the proposed system, every message m carries information used to ensure causal ordering. It uses vector clocks (Section 2.2.1.2) instead of causal barriers (Section 2.2.3) because the latter keep only direct dependencies. Therefore, causal barriers limit the amount of messages that could be bundled, reducing the performance of the system. For instance, let's consider messages m_0, m_1 and m_2, such that $m_2 \rightarrow m_1 \rightarrow m_0$, are received at node i at the order m_0, m_1, and finally m_2. The three messages must be sent by node i to the same node j. If the proposed causal broadcast aggregation protocol used causal barriers, after receiving m_0, node i would wait only for m_1 before bundling and sending to node j, because m_2 is an indirect dependency of m_0. As a result, the destination node j would have to wait for m_2 before delivering the messages in causal order. On the other hand, as shown in Section 3.3.3, with vector clocks it is possible to obtain more information about the causal history of a message due to transitive causality and, by using it, in the example, node 0 would wait for message m_2 before sending a bundled message to node j.

Every message m contains the identifier of its source ($m.s$) and the vector clock of the source at the moment of the broadcast ($m.vc$). Every node i keeps the following local variables:

- *vector_clock*: vector of size N used to store information about delivered messages;
- *vector_max*: vector of size N that keeps information about messages that can be forwarded. Each entry l of the vector keeps the sequence number of the last received message m' from l, such that all messages sent by l that precedes m' have also been received;
- *pending*: the set of messages which were received but have not been delivered yet.

Algorithm 2 presents, in details, the proposed causal broadcast protocol. It can be interpreted according to the following stages:

4.4.1 Broadcast

When node i wants to broadcast message m, it calls the function CO_Broadcast(m) (lines 6-13), which increments i's own entry in the local vector clock (line 7), assigns the identifier of i and the value of its local vector clock to m, delivers m to itself, and forwards a new packet containing m (represented as m) to i's $\log_2 N$ children.

4.4.2 Reception

Due to aggregation of messages, a node receives a packet representing a set with one or more messages sent by its parent j in the tree (line 14). Each message m in the packet is handled

Algorithm 2 Causal aggregation broadcast at node i

```
1: Init
2:     ∀l ∈ [0, N − 1] :
3:         vector_clock[l] ← 0
4:         vector_max[l] ← 0
5:     pending ← ∅

6: procedure CO_Broadcast(message m)
7:     vector_clock[i] ← vector_clock[i] + 1
8:     vector_max[i] ← vector_clock[i]
9:     m.s ← i
10:    m.vc ← vector_clock
11:    CO_Deliver(m)
12:    for all k ∈ Children(i, log2 N) do
13:        Send({m}) to k

14: upon receive mSet from j
15:    for all m ∈ mSet do
16:        pending ← pending ∪ {m}
17:        while (∃ m′ ∈ pending | m′.vc[m′.s] = vector_max[m′.s] + 1) do
18:            vector_max[m′.s] ← vector_max[m′.s] + 1
19:        for all k ∈ Children(i, Cluster(j, i) − 1) do
20:            agg ← CheckAgg(k, m)
21:            if agg ≠ ∅ then
22:                Send(agg) to k
23:    CheckDelivery( )

24: function CheckAgg(node k, message m)
25:    agg ← ∅
26:    for all m′ ∈ pending | k ∈ Children(m′.s, log2 N) do
27:        if m′.vc[m.s] ≥ m.vc[m.s] and ∄l : ( m′.vc[l] > vector_max[l]
                                               and k ∈ Children( l, log2 N ) ) then
28:            agg ← agg ∪ {m′}
29:    return agg

30: procedure CheckDelivery( )
31:    while ( ∃ m′ ∈ pending | ( (m′.vc[m′.s] = vector_clock[m′.s] + 1)
                                  and (m′.vc[k] ≤ vector_clock[k], ∀k ≠ s) ) ) do
32:        CO_Deliver(m′)
33:        vector_clock[m′.s] ← vector_clock[m′.s] + 1
34:        pending ← pending ∖ {m′}
```

independently by the receiver i and included in the *pending* set (line 16).

Node i keeps track of message receptions from each other node, by maintaining the vector *vector_max* (lines 17-18).

4.4.3 Aggregation / Forwarding

Considering the reception of m, i calls, for each of its child k in regard with m's spanning tree (line 19), the function CheckAgg(k, m) (lines 24-29) in order to aggregate all the messages in *pending* (including m), which do not have any missing pending causal precedence related to messages that must be sent to k by i (i.e., i is the parent of k with respect to the spanning tree of these messages): for every pending message m' where i is the parent of k in the spanning tree of m' (line 26), if m precedes m' (first condition of line 27) and i received all dependencies of m' to which i is responsible to forward to k (second condition of line 27), m' is added to the *agg* set. Otherwise, the forward of m' (which can be equal to m since the latter was added to the *pending* set) is postponed. If not empty (line 21), the set of aggregated message, which can be just m in the case of no possible aggregation, is then sent to k (line 22).

Considering $m_{i,j}$ the j^{th} message broadcast by node i, the *VCube* of Figure 2.6 (page 16), and spanning trees equal to the ones depicted in Figure 4.2(b,c), let's suppose that node 2 broadcasts 3 messages and node 0 broadcasts 1 message such that: *broadcast* $m_{2,1} \rightarrow$ *broadcast* $m_{2,2} \rightarrow$ *broadcast* $m_{2,3} \rightarrow$ *broadcast* $m_{0,1}$ and all messages have been received by 4, except $m_{2,2}$. In this case, for node 4, $vector\_max[2] = 1$ even if $m_{2,3}$ was received. Since $m_{0,1}.vc[2] = 3$, the conditions of line 27 are satisfied only to $m_{2,1}$, that will be sent to node 5. On the other hand, upon reception of $m_{2,2}$, $vector\_max[2] = 3$, the conditions will be true for $m_{2,2}$, $m_{2,3}$, and $m_{0,1}$ which will be aggregated into a single packet and sent to node 5. Such an aggregation takes place only for node 5. Node 4 directly sends m_0 to node 6 upon its reception.

The first condition of line 27 of CheckAgg(k, m) function is necessary in order avoid sending twice the same message. Let's take a second example where *broadcast* $m_{2,1} \rightarrow$ *broadcast* $m_{0,1}$ $\rightarrow$ *broadcast* $m_{0,2}$ and that node 4 receives $m_{0,1}$, $m_{0,2}$, $m_{2,1}$ in this order. Upon reception of $m_{0,1}$, node 4 forwards it to 6 but not to 5 and, thus, $m_{0,1}$ is held in ($pending = \{m_{0,1}\}$). The same happens upon reception $m_{0,2}$ ($pending = \{m_{0,1}, m_{0,2}\}$). However, if the first condition was not included in line 27, $m_{0,1}$ would be sent again to node 6 since the second condition is satisfied. Upon reception of $m_{2,1}$, node 4 will send the 3 messages aggregated into a single one to node 5.

Lastly, for each message $m' \in pending$, i delivers all messages whose delivery is possible following the reception of m (function CheckDelivery, lines 30-34). A message can be delivered provided that the two conditions for ensuring causal delivery order using vector clocks (Section 3.3.2) are satisfied. Once a message is delivered, it is removed from *pending*. Note that the delivery of one message can trigger the delivery of other messages. This explains why all current remaining messages in *pending* are re-checked until no more message is delivered (line 31).

4.5 Experimental Results

In this section, the proposed protocol is evaluated through simulations and the obtained results are compared to a version of the protocol without the aggregation feature.

4.5.1 Simulation Setup

The proposed causal aggregation broadcast algorithm was implemented on the top of an event-driven Java framework for simulation of peer-to-peer system called *PeerSim* (Montresor and Jelasity, 2009). The choice of such a simulation environment is due to its extensive use in different works found in the literature. Some of them are highlighted in Table 3.3 (page 40) of Chapter 3.

For the simulations, it is considered the packet-switched network delay model presented by Kurose and Ross (2012), where each packet sent by a node to another consumes $t_{pc} + t_q + t_t + t_{pp}$ units of time (*u.t.*):

- t_{pc} accounts for the processing time of a message by a node, e.g., checksum verification, aggregation and routing decisions;
- t_q is the time a message must wait in the sending queue (buffer) before being transmitted;
- t_t is the time necessary to transmit all bits of the packet to the link;
- t_{pp} expresses how long it takes for a packet to transverse the link and reach the destination node.

It is also assumed that there is no broadcast mechanism available in the system. Thus, if a message is sent to multiple destinations, a copy of it is inserted in the sending queue for each of the destinations.

Each packet has a maximum transmission unit (MTU) of 1500 bytes, where 20 bytes represent the packet header (the minimum value used by the Internet Protocol (Postel, 1981)). The size of a message was set to 50 bytes, similarly to the payload size of control messages or messages carrying monitoring information. Therefore, as messages are gradually aggregated into a packet, the current size of the packet can reach MTU size without aggregating all messages. In this case, the protocol sends the packet and continues to aggregate the missing messages in new packets, always respecting MTU size.

The number of nodes N vary from 8 up to 1024, in a power of two, and no assumption is made about the mapping between physical and logical topologies. Nodes broadcast a new message in random time given by a Poisson distribution with interval rate $\lambda = 1000$ *u.t.* while the propagation time t_{pp} of a message follows a normal distribution with mean value $\mu = 100$ *u.t.* and standard deviation $\sigma = 25$ *u.t.* Still, based on Ramaswamy et al. (2004), $t_{pc} = t_t = 1$ *u.t.*, whereas the time a message stays queued (t_q) is a function of the rate of incoming/outgoing messages and can vary for each message. Each simulation was executed 30 times and their average values are presented.

The following metrics are considered for evaluation of the results:

- *Number of packets:* the overall number of packets exchanged between nodes;
- *Number of messages per packet:* maximum number of messages that nodes aggregate into a single packet;

- *Size of packets:* size of the packet header plus the size of each composing message, whose size is given by its vector clock plus the payload itself;
- *Reception latency:* the time a message takes from its broadcast till it is received by a node;
- *Delivery latency:* the time a message takes from its broadcast till it is delivered by a node;
- *Number of buffered messages:* number of messages, received by a node, which are held in before being delivered to the application.

Without loss of correctness in capturing causal order, Birman's compression algorithm, introduced in Section 3.3.3, was used for implementing logical clock. By using this approach, when broadcasting a new message, instead of including in it the N entry values of its current vector clock, a node includes just the values of those entries that have been modified since the last broadcast by the node. Nodes continue to have information about direct and indirect dependencies of received messages.

4.5.2 Number of Packets

A straightforward consequence of message aggregation is the reduction in the number of packets that transverse the links. In the simulations, for each execution, each node broadcasts one message (i.e., all the nodes are source). Hence, for a system with N nodes with no aggregation, the total number of packets sent over the network is $N \times (N - 1)$, as each spanning tree has $N - 1$ links. The number of packets is also supposed to have an impact on other metrics, such as header size and the time a packet waits before being transmitted.

Table 4.1 shows the number of sent packets with and without aggregation, where "% of aggregation" represents the percentage of all packets that, when using aggregation, have more than one message. With aggregation, the greater the number of source nodes, the longer the paths and the higher the number of different paths (due to the organization of the trees), path intersections, and the possibility of causal relation between messages. Therefore, the number of message aggregations increases as well, leading, for instance, to in average 28.8% less transmissions ("% of reduction" in the table) with 1024 nodes.

Table 4.1: Average number of sent packets.

Nodes	***No aggr.***	***Aggr.***	***% of reduction***	***% of aggregation***
16	240	232	3.33	3.02
32	992	919	7.36	5.77
64	4032	3513	12.87	9.05
128	16256	13759	15.36	11.04
256	65280	49262	24.54	15.41
512	261632	191528	26.79	16.70
1024	1047552	745943	28.79	19.14

Even if a great number of messages are not combined to others into a packet, the percentage of aggregation causes a substantial reduction in the number of packets traveling through the

network, specially with 1024 nodes. This fact impacts other metrics, as discussed hereafter in this section.

Another interesting metric to evaluate is the number of packets that actually contain more than one message. The last column of Table 4.1 shows that from 3% up to 19% of the packets have more than one message while Table 4.2 gives, for a scenario with 256 nodes, the percentage of the overall packets (second column) that have a given number of messages. As it is possible to observe in Table 4.2, 84.59% of the packets have just one message and those with two and three represent 9.16% and 3.12% of the transmitted packets, respectively. On the other hand, only 1.2% of all packets carries more than 5 messages (up to the limit where four packets have aggregated 15 messages each). Differently, considering only the distribution of the packets with more than one message (third column), those with two or three messages account for almost 80% $(59.46 + 20.25)$ of these packets.

Table 4.2: Distribution of the number of messages per packet for a 256 node scenario.

Messages per packet	***% of all packets***	***% of aggregated packets***
1	84.59	Not applicable
2	9.16	59.46
3	3.12	20.25
4	1.26	8.18
5	0.66	4.31
(5,15]	1.2	7.8

4.5.3 Size of Messages and Packets

Besides the 20-byte header of a packet, every message included in the packet is associated with a vector clock. By applying Birman's compression algorithm in order to reduce the size of the vector clock, a message sent by i includes only the tuples $(k, vector\_clock[k]), 0 \leq k < N$ such that $vector\_clock[k]$ has changed since the last broadcast of i. Each modified entry is represented by 4 bytes (2 per item of the tuple).

In the following, the size of messages' vector clocks and the number of bytes sent over the network in a scenario with 256 nodes were evaluated. Remember that since the size of a packet is bounded to 1500 bytes, a set of messages that can be bundled together may require more than one packet (each one with a 20-byte header). Message size is given by its payload (50 bytes), source's identifier (2 bytes), and vector clock (4 bytes per entry).

Initially, when it comes to messages, each of them has its size heavily affected by its number of causal dependencies. Considering the aggregated approach, Figure 4.3 shows the percentage of messages whose vector clock carries a given number of dependencies. With no aggregation, the simulation presented the same behavior with a variation of up to 2.73% in the results.

In Figure 4.3, it is possible to observe that 27% of the messages have no causal precedence, i.e., each of them carries only its own entry in the vector clock. However, despite the small size of their respective vector clocks, these messages cannot be aggregated by our approach since the latter only combines causally related messages. For the remaining messages, 28.5% (resp., 16.8%) of them contain no more than 4 (resp., 9) causal dependencies, and this percentage continues

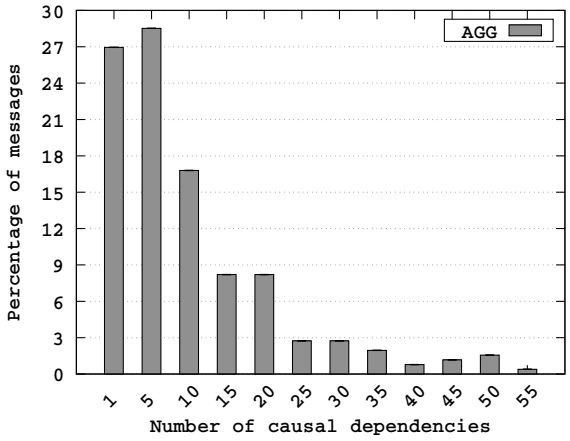


Figure 4.3: Distribution of the number of causal dependencies of a message in a scenario with 256 nodes with aggregation.

to drop till only 1 message which depends on 54 others. As each entry requires 4 bytes, 23% of the messages (those with more than 12 causal dependencies) spend more space for storing vector clock entries than the actual data from the application whose size is 50 bytes. On the other hand, the greater the number of vector clock entries, the more information gathered about causal order, which can result in more message aggregation.

Table 4.3: Distribution of the packets according to their size in bytes, for a scenario with 256 nodes.

Size (bytes)	***% of packets (No Agg.)***	***% of packets (Aggr.)***
< 100	63.67	52.66
(100,200]	30.30	31.28
(200,300]	6.03	7.62
(300,400]	0.00	3.91
> 400	0.00	4.54

Another consequence of message aggregation is the reduction in the overall number of bytes sent through the links. Since each packet has a 20-byte header, the greater the number of packets, the greater the number of headers. With no aggregation, every message is sent individually while with aggregation, the header of a packet is "shared" by several messages. Thus, the variation in the size of the packet when no aggregation is used is due only to the number of causal dependencies of its single message.

Table 4.3 shows the distribution of different packet sizes in the same 256 node scenarios with versions of the protocol without (No Aggr.) and with (Aggr.) support to aggregation. Every packet with less than 136 bytes contains necessary only one message, since in order to store two messages it is necessary, besides the payloads (100 bytes), one message with at least one causal dependency (4 bytes), vector clock entry for the source of each message (2×4 bytes), *id* for the source of each message (2×2 bytes), and the packet header (20 bytes).

The main observation from the table is that when aggregation is used by the protocol, 8.45% (3.91 + 4.54) of the packets are bigger than all packets when aggregation is not used. With aggregation there is also a reduction in the number of packets of small sizes ($<$ 100 bytes),

specially because some messages which would be sent alone are grouped with others causally related ones into a single packet. Closer to the maximum packet size, aggregation presents only 0.29% of the packets with more than 1400 bytes. The reason for this low percentage is that a packet is forwarded whenever it is not possible to include one more message in it due to lack of space, which happened to 117 out of 49262 packets in the simulation analyzed in the table.

4.5.4 Reception and Delivery Latencies

The reduction in the number of sent packets also impacts latency. Thus, two different latency metrics are considered for evaluation:

- ***Reception latency***: the time interval comprised from the broadcast of the message until it arrives at the destination node;
- ***Delivery latency***: reception latency plus the *queuing time*, i.e., the additional time a message is held in at the destination node from its reception time until it is delivered to the application.

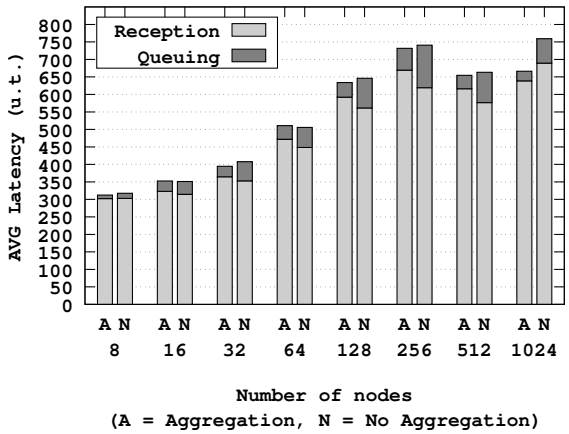


Figure 4.4: Average reception and delivery latencies.

For the experiments, every node broadcasts one message at a time chosen using a Poisson distribution. Figure 4.4 depicts the average reception and delivery latencies with and without aggregation. A first observation concerns the variation in the reception latency when the number of nodes increases. Even if the number of nodes increases 128-times (from 8 up to 1024), the average reception latency is just 2.1 times higher with aggregation and 2.2 times without it (maximum increase of 3.9 and 4.1 times, respectively). This near-logarithmic behavior can mainly be explained by the use of spanning trees to broadcast messages.

As expected, the postponement of the forwarding of messages whose dependencies are missing leads to higher reception latencies. Therefore, the same figure shows that reception latency is higher with aggregation when compared to no aggregation (except for 1024 nodes) and, as the number of nodes increases, so does the average reception latency. Furthermore, as discussed in Section 4.5.2, aggregation rate increases with the number of nodes. For instance, with 256 nodes,

aggregation poses a reception latency in average 8.1% higher compared to the same scenario with no aggregation. The different behavior with 1024 nodes is related to the number of messages that the system must deal with. In such a scenario, average reception latency with aggregation is 7.4% smaller because the average time of message forwarding postponement becomes smaller than the overhead in time necessary to send packets containing a single message. On the other hand, with no aggregation, packets stay in average 53.4% more time in the sending queue before their sending request is processed. Such a waiting time is around 50 *u.t.*, which is compliant with the difference in the same figure for the reception latency of the two approaches with 1024 nodes.

Relating to delivery latency, the results of the figure confirm our statement that delaying the forwarding of causal related messages does not degrade delivery latency but, actually, reduces it when compared to no aggregation. In networks up to 512 nodes, delivery latency difference with and without aggregation varies up to 3.2% (32 nodes), explained by the normally distributed t_{pp} (propagation time per hop). Hence, the only difference in time that aggregated messages can suffer from when compared to no aggregated messages is related to propagation or queuing times of the packets which contain them. In the scenario with 1024 nodes, there exists a greater difference between the two approaches: our causal aggregation broadcast delivers messages in average 12.2% faster. The reason is that, as previously discussed, messages are received faster with aggregation, and possibly several messages in one packet.

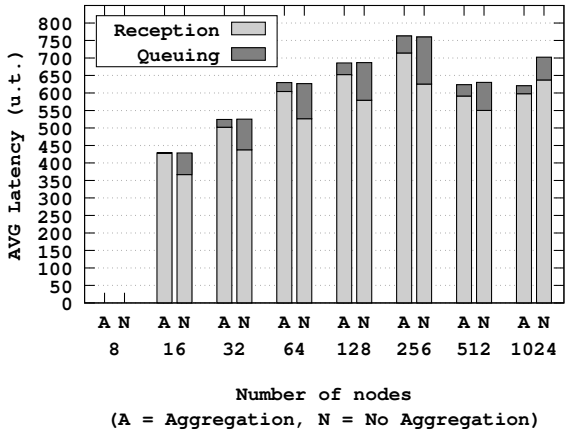


Figure 4.5: Average reception and delivery latencies for messages of aggregated messages

Another remark about Figure 4.4 concerns the *queuing* time that messages are held in (*pending*) before being delivered to the application. This waiting time is an effect of causal order but it is linked to the space a node will use to store messages as well. It can pose a bottleneck if the incoming rate of messages is high and so is the average waiting time. Regardless the number of nodes, messages are, in average, held in longer with no aggregation than with aggregation. This difference ranges from 5% (16 nodes) up to 53.4% (1024 nodes) since, with aggregation, upon the reception, more messages can be delivered immediately, reducing, therefore, the time messages are held.

It is necessary to remark that Figure 4.4 considers all messages no matter whether they

were aggregated with others or not. Thus, in order to profile the impact of message aggregation in latency, for different network sizes, Figure 4.5 considers only those packets that have two aggregated messages and compared them to the individual transmissions of the corresponding messages without aggregation, in exactly same scenarios. In Figure 4.5, for networks with 8 nodes, there is no message aggregation. However, for other network sizes, delivery latencies are the same (except for 1024 nodes for the reasons discussed before) since, for the aggregation approach, the reception latency increases (in average up to 13.6%) but the delay (*queuing* time) to deliver a message decreases (59% for 512 nodes, reaching up 74% for 64 nodes).

4.5.5 Distribution of Pending Messages

By having different trees for each source node, the proposed causal aggregation broadcast protocol does not present the bottleneck imposed by a single-rooted approach. However, before forwarding messages, every node also stores them temporarily. After a message is received at a node, it is possible that it will stay in buffer for a given time until its causal dependencies are satisfied, i.e., until all its causally preceding messages are delivered. If only a few nodes participate in the process of aggregating messages, such nodes could become a new type of bottleneck to the system, in terms of number of pending messages and capacity of processing/sending them.

Figure 4.6 shows the number of buffered messages by each node (*pending* set of Algorithm 2) in a scenario with 1024 nodes, each of them broadcasting one message. With aggregation, more than 50% of the nodes (569) buffer at most only 50 messages, while with no aggregation the ratio drops to less than 25% of the nodes. On the other hand, there exist only 24 nodes, in the aggregation case, and 212 nodes, without it, that keep at some moment more than 250 messages, Such a difference is due to the aggregation approach that avoids unnecessary buffering.

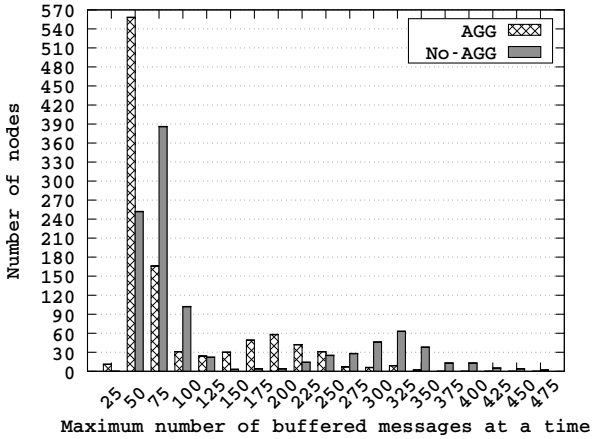


Figure 4.6: Distribution of the maximum number of messages buffered per node, with and without aggregation.

It is necessary to take into account that even if aggregation exploits only necessarily buffered messages, there exists a processing cost for carrying it out. In Figure 4.7, it is shown how each node collaborates in the aggregation process in a simulation with 1024 nodes, for two message sizes: 50 and 5 bytes. For both sizes, the distribution of the maximum number of messages

aggregated in a packet per node seems to follow a normal distribution although, for the 5-byte size, there are more nodes which have aggregated a higher number of messages per packet. The reason for such a difference is the limitation in the number of messages that a packet can hold: the smaller the size of the message, the greater the number of messages that it can keep. Every node participates in the aggregation process and most of them with a close maximum aggregation size. For the 50-byte size, 95.5% of the nodes have aggregated at some moment between 7 and 11 messages while no node has aggregated more than 14 messages. For the 5-byte size, 833 nodes (81.3%) have aggregated between 8 and 12 messages.

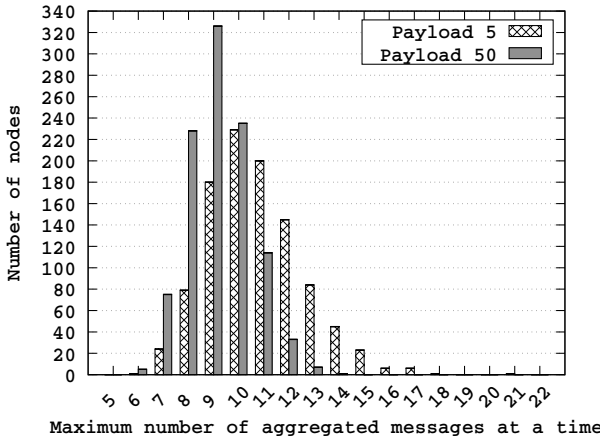


Figure 4.7: Distribution of the maximum number of messages aggregated per node for messages of size 5 and 50 bytes.

4.5.6 One Tree Versus Multiple Trees

The approach of building spanning trees rooted at the broadcast source node presents better scalability when compared to single rooted approaches like (Kim et al., 2010; Wang et al., 2012), that organize the nodes of the system in a single static distributed spanning tree. The latter has the drawback that the root can become a bottleneck since all message broadcasts start from it.

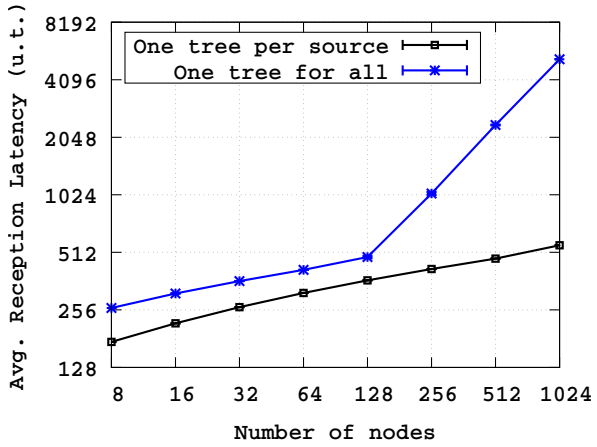


Figure 4.8: Reception latency for systems with different number of nodes using two different tree approaches: one single tree or multiple trees (one per source node) (de Araujo et al., 2017).

In Section 2.4.1, in order to justify the choice of using one tree per source instead of a single tree, it was discussed the existence of a bottleneck when a unique tree is used and its negative impact on latency. Figure 4.8 shows a comparison between a single rooted approach and another with one tree per source. All nodes of the system broadcast messages at a given rate. In the case of a single tree, for a number of nodes greater than 128, the root of the tree starts queuing messages because it cannot process all of them as fast as the input broadcast request rate. On the other hand, if each node has its own broadcast tree, the load of messages is better distributed among the nodes. Furthermore, reception latency scales well for an increasing number of nodes/messages.

Some simulations were also conducted in order to show how causal aggregation performs when all nodes broadcast messages through the same single spanning tree rooted at node 0.

In order to simulate out-of-order message receptions, latency at the links of the spanning tree of node 0 varies each time a link is used, following a Gaussian distribution, as if messages took differently routes at each broadcast. The first remark is that node 0 becomes a bottleneck and reception latencies have the same behavior of Figure 4.8. The average number of aggregated messages and number of delayed messages (in buffer waiting for delivery) for both approaches (*unique* and *multi* trees) were also evaluated. The results are gathered in Figure 4.9. With a unique tree, there are at least 86% (resp., 84%) fewer aggregations (resp., delayed messages), performed by 32 nodes (resp., 128 nodes). Such results confirm that our aggregation approach performs better with one tree per source compared to a single one because the former naturally exploits existing delays induced by different paths. The smaller number of aggregations for single tree is due to the fact that out-of-order message receptions at nodes is limited, in this case, only to latency variations of the common links over which all messages travel, which also justifies the reduced (*queuing*) time messages are held in before delivery. For unique tree with 1024, the number of aggregations decrease due to contention in the root of the tree.

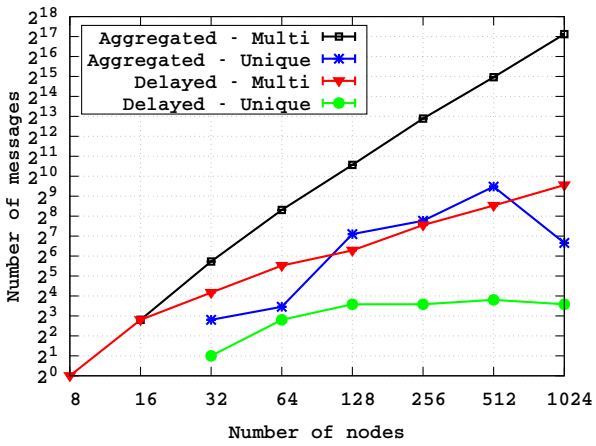


Figure 4.9: Number of aggregated and delayed messages with one spanning tree per source and a single rooted spanning tree (logarithmic scale).

4.6 Conclusion

Variations in the transmission time of messages through the links and/or network congestion can lead to a problem called *triangle inequality violation* (TIV), in which indirect transmission is faster than direct one. The protocol presented in this chapter exploits both the TIV problem and intersecting paths of per-source spanning trees built on top of *VCube* to reduce traffic and message size overheads. Therefore, a node may buffer received messages which are out of causal order and forward them to common child (or children) only when they become deliverable. Such an aggregation mechanism does not induce any overhead since the sending of a message to a child node is worthless if the latter will not be able to deliver it upon reception due to causal delivery order violation.

Results from experiments implemented on top of *PeerSim* show that the proposed aggregation protocol reduces packet traffic while keeping average delivery latency. Moreover, average buffer use is also reduced, because with the causal aggregation approach more messages are received ready to be delivered.

Chapter 5

VCube-PS: A Topic-based Publish/Subscribe System

Contents

5.1 Introduction

As discussed in Section 2.5, even if topic-based Pub/Sub systems are simpler when compared to content-based ones, they are widely applied by popular applications and services including

Twitter, Firebase/Google Cloud Messaging[1], IBM MQ[2], Apache Kafka[3], distributed multi-player online games, chat systems, mobile device notification frameworks, etc.

Concerning the construction of topic-based Pub/Sub systems, several of them found in the literature are based on per topic broadcast trees built over P2P DHTs (see Section 3.5). They use a single multicast tree which is associated to each topic composed by both subscribers (resp., brokers) and relays, i.e., non-subscribers (resp., non-brokers) of the topic that are in the logical path between subscribers. Therefore, all published messages related to a topic are broadcast through the same tree. Throughout this chapter, this kind of approach is called *SRPT* (*Single Root Per Topic*). As these systems are built over P2P DHTs, they are scalable in terms of the number of subscribers. However, relay nodes present in *SRPT* induce a higher latency and the root of a tree can become a performance bottleneck, in the case of skewed distribution of messages per topic.

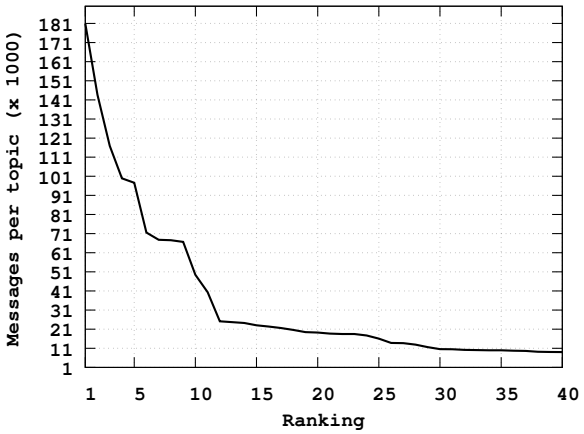


Figure 5.1: Ranking of popular topics (*hashtags*) in Twitter according to Sanli and Lambiotte (2015).

Such a skewed distribution of messages per topic was studied by Sanli and Lambiotte (2015), where the authors show that in applications like Twitter most of the publications are concentrated in few topics. Experiments show that roughly 83% of the more than 200,000 considered topics (*hashtags*) have up to 5 published messages and only 0.15% of the topics ("hot topics") are related to more than 1,000 publishing messages. The behavior of this distribution of messages is shown in Figure 5.1. Another example of such applications are multi-player online combat games where locations are mapped to topics (Gascon-Samson et al., 2015; Arantes et al., 2010). During the game, players move towards the same location increasing the publishing load for the topic corresponding to the location, i.e., the location becomes a "hot topic". Figure 2.9 (19) showed that *SRPT*-based systems may not be suitable for handling high publishing loads to the same root node due to contention constraints, i.e., the single root node broadcasts publications at a lower rate than the requests it receives to broadcast them.

Many applications that apply topic-based Pub/Sub also require that the delivery of publica-

[1]https://firebase.google.com/products/cloud-messaging/
[2]https://www.ibm.com/products/mq
[3]https://kafka.apache.org/

tions to subscribers respect causal order of publication broadcast. As example, we can mention online discussion systems, where users participating in the same group must see messages in causal order. However, as it was observed in the literature, this feature is provided by few existing Pub/Sub systems.

Considering the above points, the contribution presented in this chapter is a non-DHT Pub/Sub system named *VCube-PS*, that ensures low latency, and load balancing for publishing messages. It also respects the causal delivery order of published messages of a given topic. In the absence of churn, *VCube-PS* does not present relay nodes and, in the presence of churn, relays are temporary. Furthermore, it is important to highlight that, contrarily to many *SRPT* systems, the target of the proposed Pub/Sub system is applications that present "hot topics", i.e., high concentration of messages in few topics.

Section 5.2 presents the system model considered by *VCube-PS*, followed by the description of how spanning trees are built in *VCube-PS* (Section 5.3) and of how it implements causal broadcast (Section 5.4). The algorithms of the proposed system are presented and described in Section 5.5. Finally, in Section 5.6, evaluation results from simulations conducted on *PeerSim*, comparing *VCube-PS* with two *SRPT*-like Pub/Sub systems, are presented and discussed. One *SRPT* Pub/Sub is subscriber-based (e.g., Scribe, Magnet, DRScribe) while the second one is broker-based (e.g., DYNATOPS).

5.2 System Model and Definitions

The model considered for the development of *VCube-PS* is similar to the one used in Chapter 4 for the proposal of a causal aggregation broadcast protocol. It consists of a finite set of $\Pi = \{0, .., N-1\}$ nodes with $N = 2^d$, $d > 0$, such that each node has a unique identifier (id) and nodes communicate only by message passing. A user of the Pub/Sub system corresponds to a node. Nodes are organized in a logical hypercube.

Nodes do not fail and links are reliable. Thus, messages exchanged between any two processes are never lost, corrupted nor duplicated. The system is asynchronous, i.e., relative processor speeds and message transmission delays are unbounded. However, differently from Chapter 4.1, for the proposed Pub/Sub system, the network is fully connected, i.e., each pair of nodes is connected by a bidirectional point-to-point channel and there is no network partitioning.

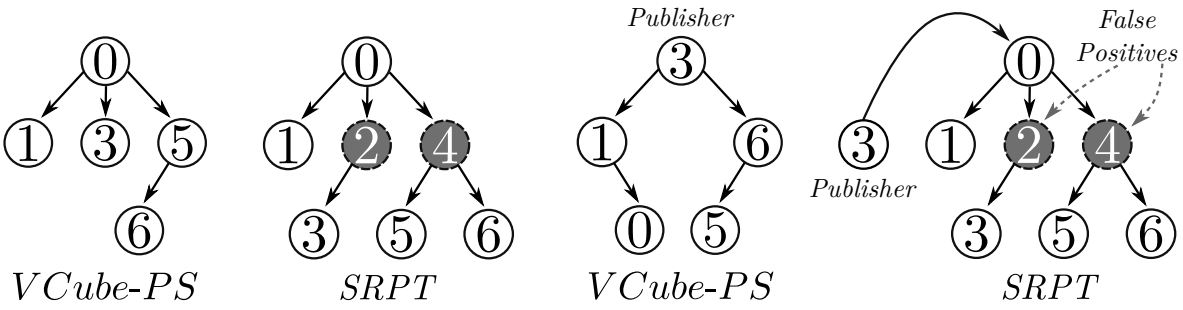
The *source* of a given message is the id of the node that publishes the message and causal delivery order is ensured for messages published to the same topic.

5.3 Per-source Spanning Trees

Contrarily to topic-based systems that use one single tree per topic, *VCube-PS* implements a different approach: multiple spanning trees are built on top of *VCube*, which is extended to cope with multiple groups (topics). Messages related to a given topic are broadcast through dynamic spanning trees rooted at the publisher composed ideally only by subscribers of the topic, i.e., when choosing its children at a given tree, a node also takes into account the subscriptions of the

child nodes. Relay nodes may take part in spanning trees, but it is only a temporary situation that happens during the propagation of unsubscription messages. Membership information (subscriptions and unsubscriptions) are also propagated using the same tree solution, thus keeping the same logarithmic properties as messages.

Figure 5.2(a) shows the difference in the trees rooted at node 0 built by *VCube-PS* and *SRPT* (also built using the same *VCube*'s topology), where a topic t is subscribed by nodes 0, 1, 3, 5, and 6. While *VCube-PS* considers only known subscribers for the construction of spanning trees, *SRPT* uses nodes 2 and 4 as relays, i.e., non-subscribers in the path between subscribers. The presence of relays in *SRPT* is equivalent to relays in DHT-based Pub/Sub (e.g., Scribe) and messages received by these nodes are called ***false positives***. Concerning the publication of messages, in Figure 5.2(b), *VCube-PS* builds a tree with lower average degree and no relay node, while *SRPT* uses an additional hop to forward messages from the publisher to the root of the topic and two false positives happen (at nodes 2 and 4).



(a) Construction of spanning trees rooted at node 0.

(b) Publication. For *SRPT*, node 0 is the root of the topic.

Figure 5.2: Example of broadcast tree for *VCube-PS* and *SRPT*. White nodes are subscribers and the gray ones are relays.

5.4 Causal and Per-source FIFO Reception Ordering

For each topic, *VCube-PS* enforces the causal order of published messages, implementing, thus, causal broadcast. It also implicitly ensures that for a single publisher, nodes will receive messages in the order they were published. It is considered that, since messages from different topics are generally not related in application level, it is not necessary to ensure causal delivery order of messages between different topics.

5.4.1 Causal Ordering

In order to implement the causal order of published messages, it is necessary to consider the subscription dynamics present in Pub/Sub systems. Subscribers may receive messages whose causal dependencies it will never receive because the latter were broadcast before the subscriber joins the topic. In this case, the subscriber must decide whether it has to wait for causal preceding messages or not before delivering a received message.

Formally, in *VCube-PS*, for the same topic t, if a node publishes a message m' after it has delivered a message m, then no node in the system will deliver m after m'. Note that if a node i never delivers m' (i.e., i leaves the topic before delivering m') or delivers m' but never delivers m (i.e., i was not subscribed to t when m was published), the causal order of published messages is not violated.

VCube-PS uses *causal barriers*, introduced in Section 2.2.3, to ensure causal delivery order. Compared to implementations of causal broadcast that use vector clocks such as Birman and Joseph (1987) (see Section 3.3.2), the key advantage of using an approach based on *causal barrier* is that the latter does not enforce the causal order based on the identifiers of the nodes but by using immediate/direct message dependencies. Thereby, causal barriers render the algorithm more suitable for dealing with the node dynamics (subscriptions and unsubscriptions) present in Pub/Sub systems.

Figure 5.3 shows an example of the application of causal barriers in *VCube-PS* to ensure per topic causal delivery order. Let's consider a distributed system with three nodes (p_0, p_1, and p_2) that have subscribed to the same topic t. Message $m_{s,t,c}$ is the message published by s with sequence number c for topic t. On the left, a timing diagram shows messages being published and delivered; the graph with message dependencies is shown on the right side. It is possible to observer that the delivery of $m_{1,t,1}$ is conditioned by the delivery of $m_{0,t,1}$ ($m_{0,t,1} \prec_{im} m_{1,t,1}$) since p_1 delivered $m_{0,t,1}$ before publishing $m_{1,t,1}$, (i.e., $cb_{m_{1,t,1}} = \{m_{0,t,1}\}$). On the other hand, $m_{1,t,2}$ directly depends on $m_{2,t,1}$ and $m_{1,t,1}$ (i.e., $cb_{m_{1,t,2}} = \{m_{2,t,1}, m_{1,t,1}\}$). Note that since $m_{0,t,1}$ precedes $m_{1,t,1}$ that precedes $m_{1,t,2}$, $m_{0,t,1}$ is an indirect dependency of $m_{1,t,2}$, and was not included, therefore, in $cb_{m_{1,t,2}}$.

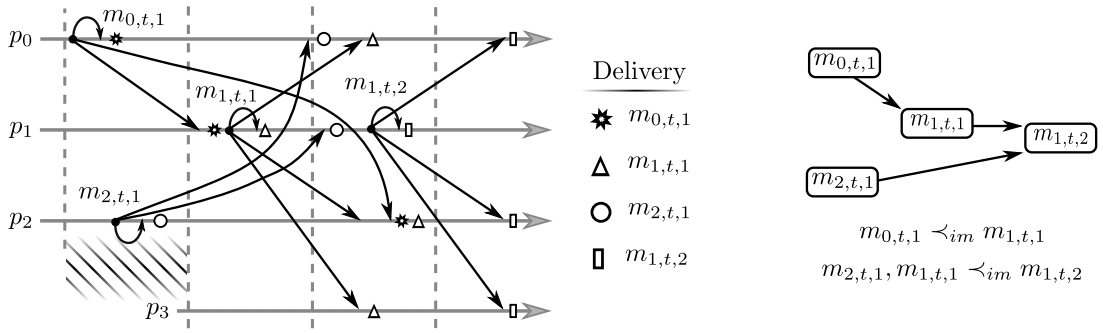


Figure 5.3: Example of causal barrier.

Now let's suppose a scenario with dynamics. In the same system shown in Figure 5.3, p_3 subscribes to t after messages $m_{0,t,1}$ and $m_{2,t,1}$ were published to the other nodes, i.e., node p_3 did not take part in the spanning trees that broadcast $m_{0,t,1}$ and $m_{2,t,1}$ and, consequently, in this case, node p_3 will neither receive nor deliver them. Let's focus particularly on the case of $m_{2,t,1}$: because it will never be received/delivered by p_3, after having delivered $m_{1,t,1}$, p_3 can deliver $m_{1,t,2}$ even if the latter also causally depends on $m_{2,t,1}$. Since nodes can dynamically subscribe to or unsubscribe from a topic in *VCube-PS*, the proposed implementation of causal order must distinguish between the case in which a message will be delivered (e.g., $m_{1,t,1}$) from the one that it will never be delivered (e.g., $m_{2,t,1}$ by p_3). To this end, *VCube-PS* guarantees per-source FIFO

reception order of messages published on a given topic.

5.4.2 Per-source FIFO Reception Ordering

Messages published by a same publisher are received by subscribers in the same order as they were produced. This order allows a subscriber of t to know that it will never receive some messages previously published, i.e., if $m'_{s,t,c'}$ is the first message that node i receives from s on topic t after it joined t's group, i will never receive $m_{s,t,c}$, $\forall c < c'$. Since no assumption can be made about the first reception of a message from a given node j, in order to avoid blocking, node i waits for messages from node j associated with topic t only after receiving a first message $m'_{j,t,c'}$. In Figure 5.3, since no message associated to t from p_2 has been received by p_3, the latter does not wait for $m_{2,t,1}$ before delivering $m_{1,t,2}$.

In *VCube-PS*, per-source FIFO reception order is ensured by the acknowledgment of published messages: a source node broadcasts a new message only after having received all the acknowledgments for the previous message it broadcast. Note that the per-source FIFO reception order is defined in regard to the reception of messages and not delivery, as in the traditional FIFO order definition presented in Section 2.3.2.

5.5 Algorithms

This section presents *VCube-PS*'s algorithms. *VCube* has been extended to satisfy *VCube-PS*'s needs. Thus, similarly to *VCube*, *VCube-PS* organizes its nodes in a logical hypercube-like topology. However, since in the used model nodes do not fail, *VCube-PS* exploits *VCube*'s organization but not its failure detection functionality. Even though it is possible to draw the analogy in which a node that has not subscribed to a topic t is considered to be faulty in relation to t. Therefore, in *VCube-PS*, the first correct node of each cluster s in $c_{i,s}$ in relation to topic t should also be a subscriber of t.

5.5.1 Types of Messages, Local Variables, and Auxiliary Functions

Each message m is uniquely identified by the source (s) and a sequence counter (c). It also carries information about the topic t. Messages can be of type `SUB` (subscription), `UNS` (unsubscribe), `PUB` (publication), and `ACK` (acknowledge). The value of the *data* field depends on the *type* of the message: for `SUB` and `UNS` messages, it holds no information while for `PUB` it carries the application message itself. In the case of an `ACK` message, if it is an acknowledge for a `PUB` or `UNS` message, the *data* field holds no information, but if it is associated to a `SUB` message, this field is used to gather membership information. `PUB` messages also carry the causal barrier (cb) for the published message.

The following local variables are kept by every node i, where MAX_TOPICS is a constant value that limits how many topics the system supports:

- *counter*: it is a local counter of node i which is incremented at every subscription, unsubscription, or publishing of a message by node i;

- $bcast\_queue[MAX\_TOPICS]$: each $bcast\_queue[t]$ is a set of pending messages (`PUB`, `SUB`, or `UNS`) related to the topic t waiting to be broadcast;
- $view[MAX\_TOPICS]$: set of the last subscription and unsubscription operations of which node i is aware. Each entry $view[t]$ has format $\langle n, o, rc\rangle$ where n is the identity of the node that has joined or left the topic t; o is equal to `SUB` or `UNS` and rc stores the value of the counter of n at the moment the subscription or unsubscription took place;
- $causal\_barrier[MAX\_TOPICS]$: each $causal\_barrier[t]$ keeps information on all messages that are immediate predecessors of the next message that will be published by node i for topic t; the causal barrier consists thus of a set of message identifiers of format $\langle s, c\rangle$ (source and sequence counter).
- $acks$: set of pending `ACK` messages for which i waits confirmation. For each message propagation to its nb children in the spanning tree of a message m identified by $\langle s, t, c\rangle$ received from j, i adds the element $\langle j, nb, \langle s, t, c, mem\rangle\rangle$ to the $acks$ set. When the `ACK` is a response for a `SUB` message, the set mem gathers membership information;
- $msgs$: set of messages that are being temporarily kept by i because they have not been delivered yet. Upon delivering m, identified by $\langle s, t, c\rangle$, the latter can be removed from $msgs$;
- $not\_delvs[MAX\_TOPICS]$: each $not\_delvs[t]$ contains a set of tuples that identify messages received by node i for topic t and not yet delivered because their respective *causal barrier* has not been satisfied. Each element has format $\langle s, c, cb\rangle$ where s is the identity of the source node that broadcast the message whose counter is c, and cb corresponds to the *causal barrier* of the message.
- $last\_delvs[MAX\_TOPICS]$: each $last\_delvs[t]$ keeps the identifiers of the last message from each publisher node delivered by node i for topic t. Each element of the set is the tuple $\langle s, c\rangle$ where s is the source identity of the message whose counter is c;
- $first\_rec[MAX\_TOPICS]$: each $first\_rec[t]$ keeps the identifiers of the first message received from each publisher for a topic t. Each element of the set is a tuple $\langle s, c\rangle$, where s is the identity of the source of the message and c is the counter of the message.

In the algorithms, the symbol $\perp$ represents a *null* element while the underscore (_) is used to indicate *any* element. Function $\#(x)$ returns the number of elements in a set x and, for a given node i, i may be used to represent the identifier of node i or the node itself, depending on the context.

The algorithm used to build distributed spanning trees extends the one presented in Section 2.4.1. Therefore, function that returns the children of a node i has been modified to take into account the topic t for which node i is building the tree:

- CHILDREN(i, t, h): A child of i is the first node of a cluster $c_{i,s}$ which is also a subscriber of topic t; or the first node in $c_{i,s}$ in case of no topic ($t =$ ' $*$ '). The parameter h can range

from 1 to $\log_2 N$. If $h = \log_2 N$, the result set contains the i's children where each child is in $c_{i,s}, 1 \leq s \leq \log_2 N$. For any other value of $h < \log_2 N$, the function returns only a subset of i's children, i.e., those children whose respective cluster number s is smaller or equal to h ($s \leq h$). For instance, in Figure 2.6 (page 15), if $t =$ ' $*$ ', CHILDREN$(0, *, 3) = \{1, 2, 4\}$, CHILDREN$(0, *, 2) = \{1, 2\}$, and CHILDREN$(4, *, 2) = \{5, 6\}$. On the other hand, if only nodes 0, 3, and 4 have joined topic t_1, CHILDREN$(0, t_1, 3) = \{3, 4\}$ and CHILDREN$(4, t_1, 2) = \emptyset$.

5.5.2 Application (User Interface) Functions

VCube-PS offers an interface consisting of functions SUBSCRIBE(t), UNSUBSCRIBE(t), and PUBLISH(t, m), all presented in Algorithm 3, which allow a node to subscribe to topic t, unsubscribe from t, and publish a message to all subscribers of t, respectively. A node can publish a message related to a topic if it is currently a subscriber of this topic (line 17). These functions generate messages of types `SUB`, `UNS`, or `PUB`, respectively, which are sent using function CO_BROADCAST (Algorithm 4) to all nodes, in case of subscription, or all subscribers of topic t, otherwise.

Algorithm 3 Functions offered as the interface to the application of every node i

```
1:  Init
2:      counter ← 0
3:      ∀t ∈ MAX_TOPICS : view[t] ← ∅

4:  function SUBSCRIBE(topic t)
5:      if ⟨i, SUB, _⟩ ∉ view[t] then
6:          view[t] ← {⟨i, SUB, counter⟩}
7:          CO_BROADCAST(SUB, t, _)
8:          return OK
9:      return NOK

10: function UNSUBSCRIBE(topic t)
11:     if ⟨i, SUB, _⟩ ∈ view[t] then
12:         view[t] ← view[t] ∖ {⟨i, SUB, _⟩}        ▷ removes subscription for t
13:         CO_BROADCAST(UNS, t, _)
14:         return OK
15:     return NOK

16: function PUBLISH(topic t, message data)
17:     if ⟨i, SUB, _⟩ ∈ view[t] then        ▷ only subscribers of t can publish at t
18:         CO_BROADCAST(PUB, t, data)
19:         return OK
20:     return NOK
```

5.5.3 Propagation of a Message

When node i invokes one of the application functions (Algorithm 3) for topic t, the procedure CO_BROADCAST (line 5 of Algorithm 4) is called, generating a new message of the corresponding type (`PUB`, `SUB`, or `UNS`) which is inserted in the queue of t. Then, a task related to t (Task

Algorithm 4 Causal broadcast algorithm and delivery executed by node i

1: **Init**
2: $\forall t \in MAX\_TOPICS$: $view[t] \leftarrow \emptyset$; $first\_rec[t] \leftarrow \emptyset$;
$not\_delvs[t] \leftarrow \emptyset$; $delv[t] \leftarrow \emptyset$; $bcast\_queue[t] \leftarrow \emptyset$
3: $msg \leftarrow \emptyset$
4: **create task** $HANDLE\_RECEIVED\_MSG$

5: **procedure** Co_Broadcast(message_type $type$, topic t, message $data$)
6: New(m)
7: $m.type \leftarrow type$
8: $m.s \leftarrow i$
9: $m.t \leftarrow t$
10: $m.c \leftarrow counter$
11: $m.data \leftarrow data$
12: $counter \leftarrow counter + 1$
13: **if** $type = SUB$ **then**
14: **create task** $START\_MSG\_PROPAGATION(t)$
15: $bcast\_queue[t].insert(m)$

16: **Task** $START\_MSG\_PROPAGATION$(topic t)
17: **loop**
18: $m \leftarrow bcast\_queue[t].first()$ ▷ block if queue is empty
19: **if** $m.type = PUB$ **then**
20: **if** $\langle i, \_ \rangle \notin first\_rec[t]$ **then**
21: $first\_rec[t] \leftarrow first\_rec[t] \cup \{\langle i, m.c \rangle\}$
22: Co_Deliver(m)
23: $last\_delvs[t] \leftarrow last\_delvs[t] \setminus \{\langle i, \_ \rangle\} \cup \{\langle i, m.c \rangle\}$
24: $m.cb \leftarrow causal\_barrier[t]$
25: $causal\_barrier[t] \leftarrow \{\langle i, m.c \rangle\}$
26: **if** $m.type = SUB$ **then**
27: $chd \leftarrow$ Children$(i, *, \log_2 N)$
28: **else**
29: $chd \leftarrow$ Children$(i, t, \log_2 N)$
30: **for all** $k \in chd$ **do**
31: Send(m) **to** k
32: **if** $chd \neq \emptyset$ **then**
33: $acks \leftarrow acks \cup \{\langle \perp, \#(chd), \langle i, t, m.c, \emptyset \rangle \rangle\}$
34: **wait until** $(acks \cap \{\langle \perp, \_, \langle m.s, m.t, m.c, \_ \rangle \rangle\} = \emptyset)$
35: **if** $m.type = UNS$ **then**
36: $msg \leftarrow msg \setminus \{m \mid m.t = t\}$; $not\_delvs[t] \leftarrow \emptyset$
37: $first\_rec[t] \leftarrow \emptyset$; $delv[t] \leftarrow \emptyset$
38: **if** $bcast\_queue[t] = \emptyset$ **then**
39: **exit**

40: **Task** $HANDLE\_RECEIVED\_MSG$
41: **loop**
42: **upon receive** m **from** j ▷ block if no message
43: **if** $m.type \neq ACK$ **then**
44: **if** $m.type = SUB$ **then**
45: $chd \leftarrow$ CHILDREN$(i, *,$ CLUSTER$(j, i) - 1)$
46: **else**
47: $chd \leftarrow$ CHILDREN$(i, m.t,$ CLUSTER$(j, i) - 1)$
48: **if** $chd = \emptyset$ **then** ▷ leaf node
49: NEW(m')
50: $m'.type \leftarrow ACK$
51: $m'.s \leftarrow m.s$
52: $m'.t \leftarrow m.t$
53: $m'.c \leftarrow m.c$
54: $m.data \leftarrow \emptyset$
55: SENDACKS(j, m')
56: **else** ▷ propagate m
57: $acks \leftarrow acks \ \cup \{\langle j, \#(chd), \langle m.s, m.t, m.c, \emptyset\rangle\rangle\}$
58: **for all** $k \in chd$ **do**
59: SEND(m) **to** k
60: **else** ▷ $m.type = ACK$
61: $k, nb, mem \leftarrow k', nb', mem' : \langle k', nb', \langle m.s, m.c, m.t, mem'\rangle\rangle \in acks$
62: $acks \leftarrow acks \setminus \langle k, nb, \langle m.s, m.c, m.t, mem\rangle\rangle$
63: $m.data \leftarrow m.data \cup mem$
64: **if** $nb > 1$ **then**
65: $acks \leftarrow acks \cup \langle k, nb - 1, \langle m.s, m.c, m.t, m.data\rangle\rangle$
66: **else if** $k \neq \perp$ **then** ▷ All pending $ACKs$ were received
67: SENDACKS(k, m)

68: **if** $\langle i, SUB, \_ \rangle \in view[m.t]$ **then** ▷ i is subscribed to $m.t$
69: **if** $m.type = PUB$ **then**
70: **if** $(\nexists \langle m.s, \_\rangle \in first\_rec[m.t])$ **then**
71: $first\_rec[m.t] \leftarrow first\_rec[m.t] \cup \{\langle m.s, m.c\rangle\}$
72: $not\_delvs[m.t] \leftarrow not\_delvs[m.t] \cup \{\langle m.s, m.c, m.cb\rangle\}$
73: $msgs \leftarrow msgs \cup \{m\}$
74: CHECKDELIVERY$(m.t)$ ▷ received messages may be delivered
75: **else if** $m.type = ACK$ **then**
76: $view[m.t] \leftarrow$ UPDATE$(view[m.t], m.data)$
77: **else** ▷ SUB or UNS message
78: $view[m.t] \leftarrow$ UPDATE$(view[m.t], \{\langle m.s, m.type, m.c\rangle\})$
79: **if** $m.type = UNS$ **then**
80: $first\_rec[m.t] \leftarrow first\_rec[m.t] \ \setminus \{\langle m.s, \_\rangle\}$

81: **function** UPDATE(view set_1, view set_2)
82: **for all** $\langle n_1, \_, rc_1\rangle \in set_1$ **do**
83: **if** $(\exists\ \langle n_1, \_, rc_2\rangle \in set_2)$ **then**
84: **if** $rc_2 > rc_1$ **then**
85: $set_1 \leftarrow set_1 \setminus \{\langle n_1, \_, rc_1\rangle\}$
86: **else**
87: $set_2 \leftarrow set_2 \setminus \{\langle n_1, \_, rc_2\rangle\}$
return $set_1 \cup set_2$

88: **procedure** CHECKDELIVERY(topic t)
89: **while** $(\exists\ \langle s, c, cb\rangle \in not\_delvs[t] :$ CHECKCB$(t, cb) = true)$ **do**
90: CO_DELIVER$(m), m \in msgs$: $m.s = s$, $m.t = t$, **and** $m.c = c$
91: $not\_delvs[t] \leftarrow not\_delvs[t] \setminus \{\langle s, c, cb\rangle\}$
92: $msgs \leftarrow msgs \setminus \{m\}$
93: $last\_delvs[t] \leftarrow last\_delvs[t] \setminus \{\langle s, \_\rangle\} \cup \{\langle s, c\rangle\}$
94: $causal\_barrier[t] \leftarrow causal\_barrier[t] \setminus cb\ \cup \{\langle s, c\rangle\}$

95: **function** CHECKCB(topic t, causal barrier cb)
96: **for all** $\langle s, c\rangle \in cb$ **do**
97: **if** $\left(\begin{array}{l}(\exists\ \langle s', c'\rangle \in last\_delvs[t]:\ s = s'\ \textbf{and}\ c' \geq c)\\ \textbf{or}(\exists\ \langle s', c'\rangle \in first\_rec[t]:\ s = s'\ \textbf{and}\ c' > c)\end{array}\right)$ **then**
98: $cb \leftarrow cb \setminus \{\langle s, c\rangle\}$
return $(cb = \emptyset)$

99: **procedure** SENDACKS(node j, message m)
100: **if** $(\langle i, SUB, \_\ \rangle \in view[m.t]$ **and** $\nexists \langle m.s, \_\rangle \in first\_rec[m.t])$ **then**
101: $m.data \leftarrow m.data \cup \{\langle i, SUB, c\rangle : \langle i, SUB, c\rangle \in view[m.t]\}$
102: SEND(m) **to** j

$START\_MSG\_PROPAGATION$) continuously removes the first message from this queue and starts the broadcast. The next message is removed from the queue only after the reception of acknowledge (message `ACK`) from all current subscribers (per-source FIFO reception order) to whom node i sent the previous message (line 34). The task associated with t is created when node i becomes a new subscriber of the group of topic t (line 14).

In Algorithm 4, task $START\_MSG\_PROPAGATION$ for topic t starts the propagation of m, the first message removed from the queue (line 18), by dynamically building a hierarchical spanning tree, rooted at i, whose composition depends on the type of the message (lines 26-31):

- `PUB` or `UNS`: when a node sends messages of these types, it is already member of the associated topic t, i.e. the node has membership information about t. Thus, the spanning tree is composed only by the subscribers of t.
- `SUB`: upon subscription to topic t, a node does not know the current set of subscribers of t. Thus, spanning trees used to propagate `SUB` message are composed by all nodes.

For this purpose, node i calls function CHILDREN$(i, t, \log_2 N)$ which renders, for `PUB` and `UNS`

messages, the set of the first subscriber nodes of t for each of its clusters (line 29) or the first node of each of i's clusters (line 27) in the case of a SUB message ($t =$ '$*$'). These nodes become i's children in the spanning tree and m is sent to them. Upon receiving m from a node j, by calling function CLUSTER (line 45 or 47 depending on the type of message), every child of node i sends m to its own children in the first $s-1$ clusters, in relation to topic t and the cluster s of j to which i belongs. These nodes then become j's children and the process continues until m is received by all leaf nodes.

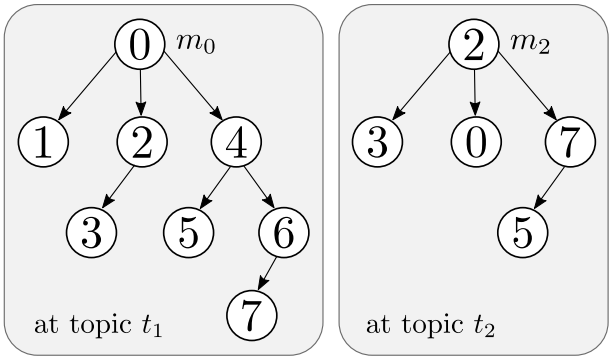


Figure 5.4: Broadcast trees for two different sources and topics.

For instance, consider the left side of Figure 5.4, that all nodes are subscribers of t_1, and that node p_0, subscriber of t_1, wants to publish a message m_0 related to t_1 (`PUB` messages). p_0 is the root of the respective spanning tree: m_0 will be sent to the $\log_2 N = 3$ children of p_0 (CHILDREN$(0, t_1, 3) = \{1, 2, 4\}$). Upon the reception of message m_0, p_1 does not forward it since CHILDREN$(1, t_1, 0) = \emptyset$, while p_2 forwards it to its child p_3, the first subscriber of cluster $c_{2,1}$ (CHILDREN$(2, t_1, 1) = \{3\}$). When p_3 receives m_0, as CHILDREN$(3, t_1, 0) = \emptyset$, p_3 does not forward m_0 to any node. However, in the case of p_4 (CHILDREN$(4, t_1, 2) = \{5, 6\}$), it forwards m_0 to its children $p_5 \in c_{4,1}$ and $p_6 \in c_{4,2}$. Finally, p_6 sends m_0 to p_7.

Consider now a second example, on the right side of Figure 5.4, where only p_0, p_2, p_3, p_5, and p_7 are subscribers of t_2 and p_2 publishes m_2 related to t_2. In this case, p_2 sends m_2 to each of its child of its $\log_2 N = 3$ clusters that are also subscribers of t_2: CHILDREN$(2, t_2, 3) = \{3, 0, 7\}$ (p_6 is the first node in $c_{2,3}$ but it is not subscribed to t_2). Upon receiving m_0, p_3 does not forward it, because it is already a leaf node in the tree. Node p_0 does not forward it to p_1 since the latter is not a subscriber of t_2. On the other hand, p_7 verifies that in cluster $c_{7,2} = (5, 4)$, p_5 is a subscriber of t_2 (CHILDREN$(7, t_2, 2) = \{5\}$), and therefore sends m_2 to p_5 which on its turn does not send it to p_4, because even if p_4 is the first and only node in $c_{5,1}$, it is not a subscriber of t_2.

After forwarding a message m to a child k, node i waits for an `ACK` message from k, which confirms the reception and propagation of m by k. In order to control pending `ACK`s, whenever node i publishes a new message to its children, it adds to the set $acks$ a tuple that identifies m and the number of pending `ACK`s it is waiting for (line 33). If i is not the source of the message (i.e., it is forwarding m in the tree), this tuple also contains the node j from which it received the message (line 57). After the reception of m, a node will send an `ACK` to its parent node only after it receives itself ACK messages from all its current children related to the topic in question (lines 64-67). `ACK` messages will, thus, be propagated to the root, the source node of m.

Eventually the latter receives all the ACK messages it waits for and, in this case, the task related to t removes the next message to be published from the queue associated to the topic t, if there is one. These sequences of `SUB`, `UNS`, or `PUB` and then `ACK` messages from/to the source ensure the per-source FIFO reception order of published messages of the topic, described in Section 5.4.2.

5.5.4 Reception and Delivery of Messages

When receiving a `PUB` message m of topic t from s (lines 69-74), if node i is a subscriber of t and has not delivered m yet, it keeps m in set $msgs$ and both its identification and causal barrier in set $not\_delvs[t]$. If m is the first message received from s to t, i registers it in $first\_rec[t]$, in order to enforce the causal dependencies even under the dynamics of subscriptions. Then, node i verifies, based on direct causal dependencies, which of the previously received messages can be delivered to the application. To this end, node i invokes the function CHECKDELIVERY(t) (lines 88-94) which, in it its turn, calls CHECKCB(t, cb) in order to check direct dependencies (line 95-98). A message m can be delivered by i only when every message m' on which m causally depends either (1) has already been delivered to i or (2) will never be received by i because *VCube-PS* has not considered i as a subscriber of t during the construction of the spanning tree that broadcast m'. In other words, for the second case, the first `PUB` message received from s on topic t by i has a higher sequence number than the sequence number of m'. Such a detection of the first message is possible thanks to the $first\_rec_i[t]$ set and the fact that, for the same source, publications of messages of the same topic respect per-source FIFO reception order.

After delivering m, node i removes it from its pending messages (lines 90-93) and updates its local causal barrier variable (line 94). Function CO_DELIVER just renders the message to the application. Note that, since the delivery of one message m can enable the delivery of other messages that causally depend on m, all remaining non delivered messages are re-checked by the CHECKDELIVERY(t) until no more message can be delivered.

5.5.5 Membership Management

In *VCube-PS*, distributed spanning trees are also used to notify membership changes. When a node i subscribes to a topic t, a broadcast `SUB` message will be received by all subscribers of t. Differently, when node i unsubscribes from a topic t, only the current subscribers of of t will receive the broadcast `UNS` message. Upon receiving either a `SUB` or `UNS` message, a subscriber of t updates its view of the membership related to t (line 78) by calling function UPDATE(view set_1,view set_2) (lines 81-87) which merges two membership sets, keeping only the current subscribers.

When a node i subscribes to a topic t, the `ACK` messages related to the `SUB` messages will also gather information about t's membership. Function SENDACKS (lines 99-102) is responsible for sending `ACK` messages. Before forwarding a received `ACK` message to its parent, each subscriber of t includes in the message its current view of t's membership (line 101) merged with the partial membership information coming from its own children (line 63). Upon receiving all `ACK` messages from its children, the new subscriber i is aware of t's membership. Figure 5.5 presents, from left

to right, an example of the ascending wave of `ACK` messages as response to the new subscription of node 0 to a given topic whose subscribers are nodes 1, 4, 5, and 6. Note that if a node is not subscribed to the topic, it has no knowledge about the topic's membership and, thus, adds no new element to the `ACK`.

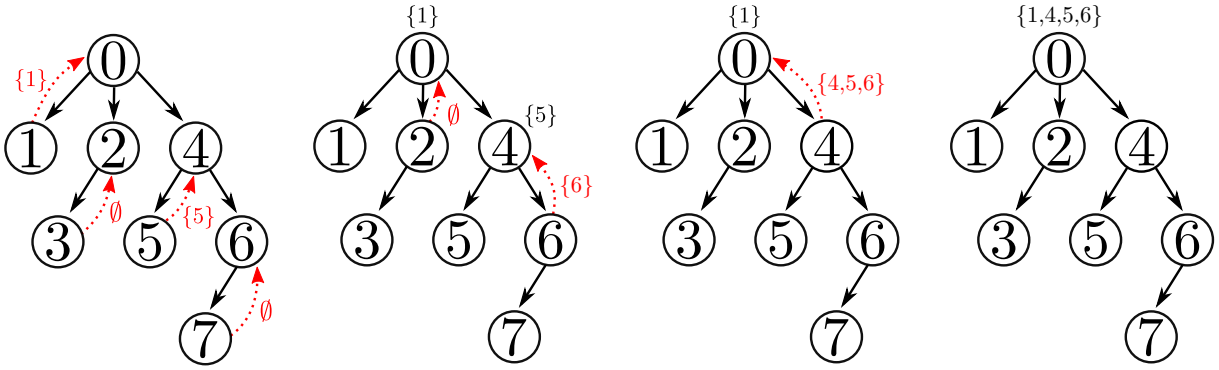


Figure 5.5: `ACK` messages sent in response to the subscription of node 0 to a topic whose subscribers are nodes 1, 4, 5, and 6.

If node i unsubscribes from topic t, it no longer delivers messages related to the topic (line 36). On the other hand, i can continue to forward messages related to t to the other subscribers of t in the spanning tree if one of the following situations occurs: (1) there exist subscribers of t that are not aware of i's unsubscription, i.e., they have not received the corresponding `UNS` message from i yet or (2) there are messages queued in i's *bcast_ queue[t]* waiting to be forwarded. Node i also sends `ACK` messages to its parent node in the respective spanning tree. These `ACK` messages are related to published messages that i received and forwarded before leaving t or to messages that satisfy the above-mentioned situations. However, eventually all `ACK` messages will be sent and, thereafter, node i will no more take part in the broadcast of messages related to t. When a subscriber of t receives an `UNS` message related to node i, it removes i from its view of t's membership (line 78) as well as the information about the first message received from i with regard to t (line 80). The latter will be renewed if i rejoins t later.

5.6 Experimental Results

In this section, *VCube-PS* is evaluated according to several metrics and, in the majority of the scenarios, compared *SRPT* approaches.

5.6.1 Simulation Setup

The simulation environment (*PeerSim*), as well the model used for describing the different components of time considered for the simulations, are the same as used in Section 4.5.1. Thus, t_{pc} (processing time) $= t_t$ (transmission time) $= 1$ *u.t.* and t_{pp} (propagation time) $= 100$ *u.t.*

It is considered that there is no broadcast mechanism available in the system. Thus, if a message is sent to multiple destinations, a copy of it is inserted in the sending queue for each of the destinations.

For most experiments, the number of nodes N varies from 8 up to 4096, in a power of two, and each experiment was executed 40 times. The following metrics are used for comparison between *VCube-PS* and *SRPT*-based approaches:

- *Latency*: the time that a published message takes to be received and delivered by all subscribers;
- *Number of messages*: overall number of `PUB` messages;
- *Number of messages to be processed by a node*: size of the queue of each node;
- *Size of `PUB` messages*: characterizes the number of direct causal dependencies that `PUB` messages hold;
- *Number of false positives*: number of messages received by nodes that act as relay nodes of messages of type `PUB`.

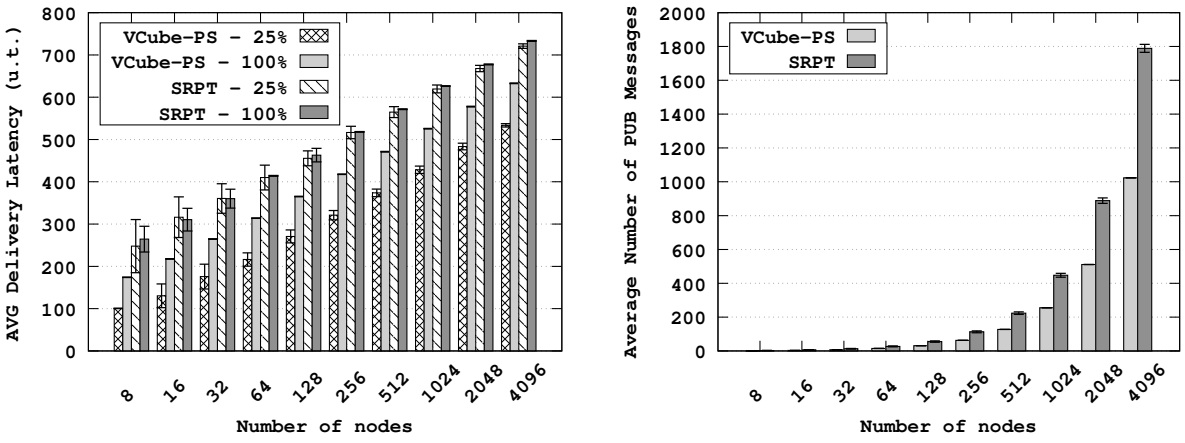
When using a *SRPT*-based system, for each topic, a node is randomly selected to act as the root of the single broadcast tree of the topic and *SRPT* trees are also built according to *VCube*'s topology.

5.6.2 A Single Publisher

This experiment evaluates the impact of the logarithmic properties of *VCube-PS*, where a single publisher publishes one message. Hence, when a subscriber receives the message, there is no delay for delivery because there is no causal ordering treatment. Figure 5.6(a) shows the delivery latency when the number of nodes of the system varies and either 25% or 100% of them are subscribers. For the first case, the set of subscribers is randomly chosen following a uniform distribution. In the case of 4096 nodes with 25% of subscribers uniformly distributed, latency in *VCube-PS* is on average 533 units of time, 26% less compared to the one presented by *SRPT* in the same scenario (720 *u.t.*). It is important to remark that when 100% of the nodes are subscribers, *SRPT* has no relay and, therefore, the latency of both Pub/Sub systems is always proportional to $\log_2 N$. The only difference in this case is that *SRPT* has an additional hop as the message to be published must be sent to the root of the single tree.

The average number of `PUB` messages follows the same behavior as shown in Figure 5.6(b). In the figure, for the two approaches with 25% of the nodes as subscribers, the number of `PUB` messages used by *VCube-PS* corresponds always to the number of nodes, since there is no relay node in the tree. On the other hand, relays in *SRPT* are responsible for, for instance, 1.79 times more messages (for 4096 nodes) compared to *VCube-PS*. As the number of nodes increases, this difference is reduced, although *VCube-PS* generates, on average, at least 43% fewer messages than *SRPT* (4096 nodes).

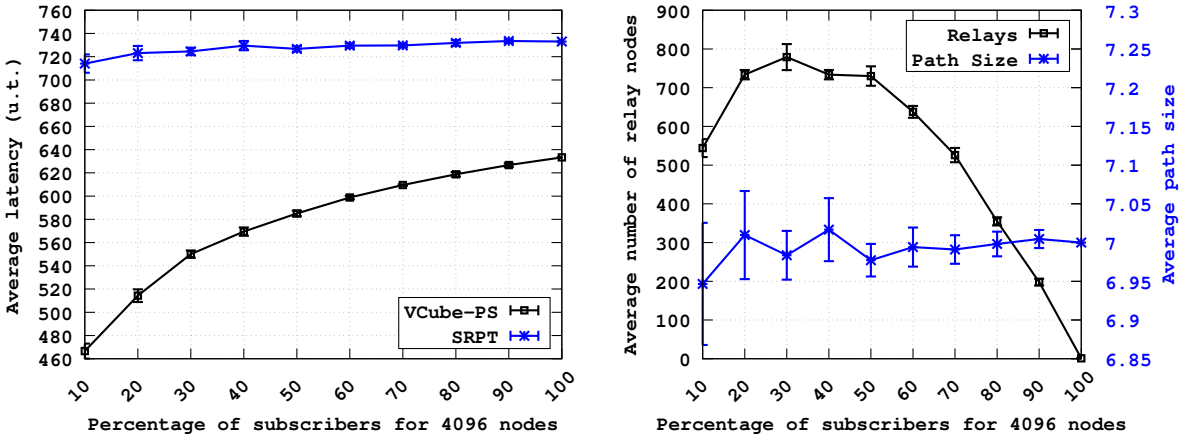
Figure 5.7 provides a more detailed analysis of the impact of the number of subscribers in *VCube-PS* and *SRPT* performance. The system has 4096 nodes and the number of subscribers, uniformly distributed, varies from 10% up to 100%. In Figure 5.7(a), *VCube-PS* performs logarithmically with respect to the number of subscribers while *SRPT* does not. A tenfold increase



(a) Average latency.

(b) PUB messages with 25% of subscribers.

Figure 5.6: Average latency and number of PUB messages for *VCube-PS* and *SRPT* with different number of subscribers.



(a) Average latency.

(b) Average number of relays and path size for *SRPT*.

Figure 5.7: The impact of different numbers of subscribers on a network with 8192 nodes.

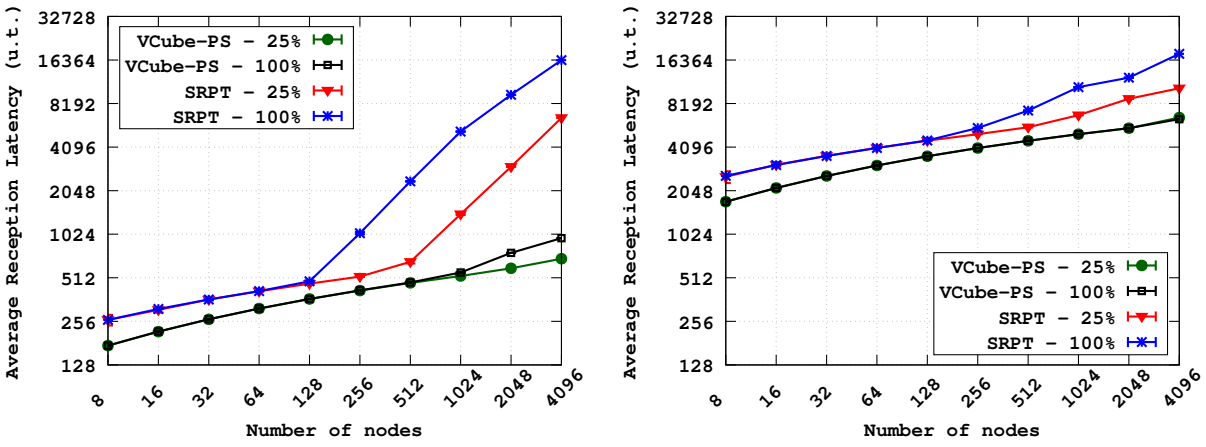
of the number of subscribers induces just a 36% increase of the average latency of *VCube-PS*. On the other hand, even if the average latency of *SRPT* varies up to approximately only 2.7%, it is always higher compared to *VCube-PS*. The minimum difference between the two approaches is observed when all nodes are subscribers, and such a difference corresponds to the additional hop used by *SRPT* to send the message to the tree root (on average close to 100 *u.t.*).

Figure 5.7(b) helps to better understand the behavior of *SRPT*. Considering all the 4096 nodes of the system, the tree can have up to 12 ($\log_2 4096$) levels. If a subscriber is a leaf node, the tree will have a branch with 12 levels, even if no other node in the branch is a subscriber. When 30% of the nodes are subscribers (i.e., around 1228 nodes), there exist, on average, 779 relays, resulting in a tree with almost 50% of the nodes of the system. However, as the number of subscribers increases, they replace relay nodes in the tree. Naturally, the number of relays tends to 0 as the number of subscribers increases. Despite of this behavior, due to the uniform

distribution of subscribers, the average path size that the message travels over the tree follows a constant pattern (around 7 hops) no matter the percentage of subscribers. Thus, the quasi-constant latency of *SRPT* in Figure 5.7(a) is related to the position of relay nodes instead of their amount, because their position affect the average path size.

5.6.3 Several Publishers

This experiment analyzes the behavior of both approaches in the presence of multiple publishers. All nodes are subscribers of a single topic and the number of publishers varies. Each publisher i sends one message at time t_i which is uniformly distributed between $[0, 1000]$ units of time. By having multiple publishers of the same topic, differences in latency will arise from the distribution of the load among the nodes when using one root per publisher (*VCube-PS*) or one root per topic (*SRPT*).



(a) 1/100 ratio ($t_{pp} = 100$). (b) 1/1000 ratio ($t_{pp} = 1000$).

Figure 5.8: Average reception latency with 25% and 100% of publishers (log. scale).

Figure 5.8 shows in logarithmic scale the average reception latency when the number of nodes of the system varies and either 25% or 100% of them are publishers. Here it is important to recall that the ratio between the processing time (t_{pc}) and the propagation time (t_{pp}) has an impact on the load contention, since it represents the difference between the input and output rates of messages. In Figure 5.8(a), it is considered a the ratio 1/100 (which is used in all other evaluations of this work) and Figure 5.8(b) uses a propagation time which is ten times greater, ($t_{pp} = 1000$ *u.t.*), leading to a ratio 1/1000, i.e. messages are less prone to contention but will, at the same time, take longer to be received by the subscribers.

In Figure 5.8(a), *VCube-PS* has a load distribution with a maximum increase of 38.8% (4096 nodes and 100% of publishers) when compared to *VCube-PS* with 4096 nodes and 25% of publishers. It happens because even though there are 4 times more messages, they traverse different paths in the network. This result corroborates that the use of one tree per publisher helps to distribute the load, since each message traverses a different path in the network. On the other hand, as *SRPT* imposes a unique tree for disseminating messages to subscribers of a topic, if several messages arrive at the root of the tree at the same time they will be queued before trans-

Table 5.1: Average size of the queue per group of nodes.

# of messages	*# of nodes (VCube-PS)*	*# of nodes (SRPT)*
0	0	512
$(0, 2]$	0	448
$(2, 4]$	0	60
$(4, 8]$	495	3
$(8, 16]$	510	0
$(16, 32]$	19	0
$(32, 4096]$	0	0
$(4096, 8192]$	0	1

mission, increasing, thus, the reception latency. For up to 128 nodes, *SRPT* latencies are on average one hop in time higher compared to *VCube-PS*, because in these cases the arrival and output rates of messages are close, leading to no contention. Beyond this number of nodes, the root receives more messages than it can process and transmit per interval of time and starts to saturate. For instance, in comparison with *VCube-PS* with 256 nodes and 100% of publishers, *SRPT* has an average latency 2.48 times greater, and this ratio grows linearly after this point.

Comparing Figure 5.8(b) to Figure 5.8(a), the average reception latency increases less in *SRPT* in relation to *VCube-PS* because, with a 1/1000 ratio, it takes longer to receive messages, although the output throughput remains the same.

Table 5.1 shows the distribution of nodes according to the average size of their sending queues, in a scenario with 1024 nodes, 1/100 ratio, and where all nodes are publishers and subscribers.

The size of the sending queue has a direct impact in the reception latency. The load distribution on the nodes in *SRPT* is uneven when compared to *VCube-PS*: 98% of the nodes in *VCube-PS* have an average load between $(4, 16]$ messages, while 44% of the nodes in *SRPT* have on average between $(0, 2]$ messages in their buffers. In *SRPT*, 50% of the nodes simply do not participate in the routing of any message, because they are leaf nodes of the single tree of the topic and one node (the root) has an average load of 9240 ($\sigma = 4617$) messages, which incurs in high reception latencies.

5.6.4 Message Order

Besides the published message itself, every `PUB` message contains its causal barrier, i.e., a list with direct causal dependencies of the published message. Thus, the size of a `PUB` message increases depending on the number of elements in this list. In order to evaluate the size of such a list and the latency due to message ordering in *VCube-PS*, this experiment considers that one node s, chosen randomly, publishes a first message m_s. Upon receiving it, each node k waits for a random time (t_w) before broadcasting message m_k, similarly to a message discussion group service where all members of the group answer publicly to a question posted by one of them. For N nodes, there will be $N^2 - N$ messages. Additionally, this scenario is extended for the case in which a node k has to wait for at least p messages before broadcasting its own. To this end, there are $p \geq 1$ nodes that independently broadcast a message, each in the beginning of the experiment. Just after receiving all these initial messages, any node can publish a message.

Figure 5.9 groups messages according to the size interval of their causal barriers for *VCube-*

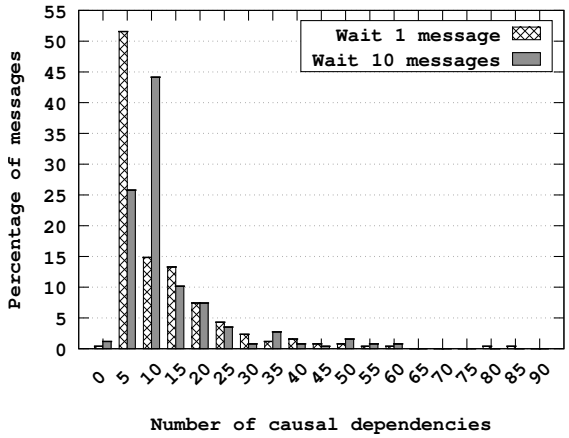


Figure 5.9: Frequency distribution for the number of causal dependencies of a message in a network running *VCube-PS* with 256 nodes.

PS. When it is necessary to wait for just one message before a node broadcasts its own message, 51.6% of the messages generated in the system have less than 5 preceding messages. More precisely, 19.9% of them have just one causal dependency. On the other hand, if a node waits for more messages (10 in the case of the figure) before broadcasting its own, a larger number of nodes will have 10 or more direct dependencies. In this case, 35.2% of the messages have size 10 (10 direct dependencies) and 79.7% of them have fewer than 15. However, in both cases, the number of direct dependencies keeps a reasonable size.

The additional delay imposed by causal barriers before delivering a message to the application was also evaluated. When a node waits for 1 message before broadcasting its own, about 95.1% of the messages are delivered in less than 10 *u.t.* after the message is received (87.2% are delivered with no delay). Only 81 messages (out of 65280) have a delay higher than 50 *u.t.*, with an upper limit of 150 units of time. Increasing the number of the waiting messages to 10, 457 messages wait more than 50 *u.t.* to be delivered (maximum 187), although the number of messages with no delay remains high (84.2%).

5.6.5 Multiple Topics

As discussed in Sanli and Lambiotte (2015), in real world applications like Twitter, a few topics are related to most of the messages. The authors show that in Twitter, roughly 60% of the topics have only one message published, 83% of them have no more than 5, only 0.15% of the topics are related to more than 1000 messages each. This behavior follows a Zipf-like distribution with a coefficient of 0.825 according to the data provided in the reference. In this experiment, *VCube-PS* and *SRPT* are evaluated in a scenario with multiple topics. Messages are assigned following both the Zipf-like and uniform distributions. Figure 5.10 depicts the results for 256 nodes, 128 topics, and a varying number of messages. Each node publishes a new message on average every 500 *u.t.* for a topic, randomly chosen. Therefore, messages are uniformly distributed among the publishers, but not necessarily among the topics.

No matter the distribution of messages among the topics, *VCube-PS* always relies on the

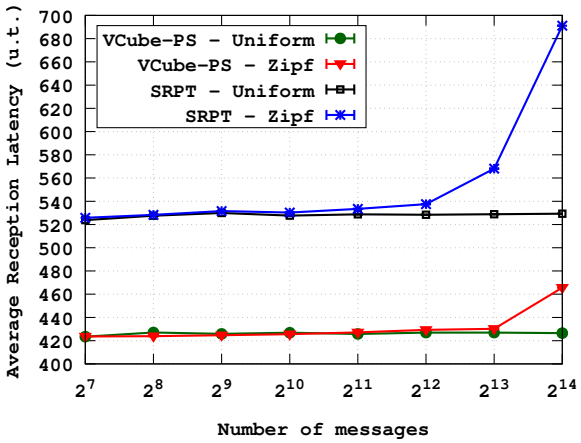


Figure 5.10: Average reception latency with 256 nodes and 128 topics for two distribution of messages per topic.

same root for a given publisher, while *SRPT* does not. This is the reason why the behavior of *SRPT* is the same as *VCube-PS*'s for a uniform distribution of messages. However, when the number of messages sent per node increases beyond a threshold, *VCube-PS* increases the latency due to contention at the source of the messages, i.e., the root of the tree. On the other hand, for the Zipf distribution, *SRPT* has an average reception latency 30.6% higher compared to the uniform distribution (for 2^{14} messages). *VCube-PS* increases latency, on average, only 9.2%.

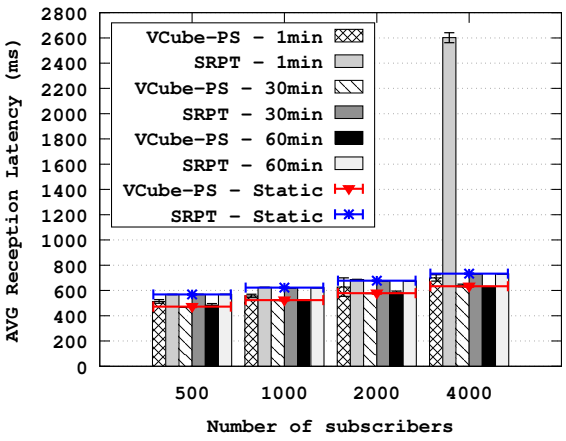
These results confirm that *VCube-PS* is scalable in terms of publishers, while *SRPT* is scalable in terms of topics. However, in real scenarios, most of the messages are concentrated on a small number of topics.

5.6.6 Churn Evaluation

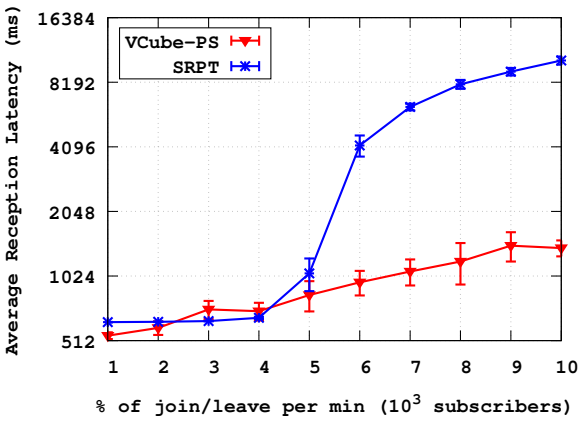
This set of experiments evaluates how *SRPT* and *VCube-PS* tolerate membership changes. The parameters used for the evaluation are those proposed by Rhea et al. (2004), which considers that the time a node stays connected to a P2P system (session time) is heterogeneous and that the average time ranges from a few minutes up to hours, following a Poisson process. For every node that leaves a given topic, another randomly selected node joins that topic, thus, always keeping the number of subscribers equals to N_s nodes[4].

For the experiments, it it assumed one topic and each unit of time represents $1ms$. Every $500ms$, a new message is published by a randomly selected node (uniform distribution). Each simulation corresponds to a network running for 120 minutes. Figure 5.11 presents the average reception latency and standard deviation. It is worth reminding that in *VCube-PS*, every membership change (subscription or unsubscription) generates a new message which is broadcast to all nodes of the system, similarly to a publishing message, while *SRPT* needs to rebuild its per topic single trees. Furthermore, *SRPT* trees often have relay nodes (non-subscriber nodes) while in *VCube-PS*, when a node i unsubscribes, it can still receive and forward publications related to the topic for a while (temporary relay node, see Section 5.5.5).

[4] N_s is smaller than the total number of nodes of the simulation in order to have a pool of candidates for new subscriptions and, at the same time, keep the same hypercube dimension throughout the experiment.



(a) Different numbers of subscribers and churn rates.



(b) 1000 subscribers and a varying churn rate.

Figure 5.11: Average reception latency under churn.

Figure 5.11(a) summarizes the results with 500, 1000, 2000, and 4000 subscribers. The dynamics of subscriptions were simulated for three different average session times (t_{med}): 1, 30, and 60 minutes. For baseline comparison sake, along with the scenarios with churn, the figure also shows results with static membership. Standard deviation values, although small, are also depicted.

Comparing *VCube-PS* with churn to the static baseline, the former presents average latencies up to 10% higher. In other words, to some extent, *VCube-PS* is sensitive to churn since static membership does not induce false-positives while, with churn, *VCube-PS* has temporary relay nodes, responsible for the 10% latency increase. On the other hand, except for 4000 subscribers, *SRPT* latencies vary only up to 1.4% compared to the corresponding static baseline. This stable behavior can be explained as, even in scenarios with no churn, *SRPT* trees have usually non-subscribers (relays) and, therefore, the size of their branches does not vary with churn. However, these relays are also responsible for the longer *SRPT* tree branches when compared to *VCube-PS* ones, justifying why, for a given churn rate, *SRPT* presents higher latency than *VCube-PS*, independently of the number of subscribers. The highest impact of the churn is observed in *SRPT* with 4000 subscribers and $t_{med} = 1min$, with approximately 46 unsubscriptions and 46 new subscriptions per minute. In this case (high churn rate), the average latency is much higher than the static one (3.56 times), not only because of the presence of false-positives (2.74% of all received `PUB` messages), but also due to contention caused by `SUB` and `UNS` messages. A last interesting observation is that, except for *SRPT* with 4000 subscribers and $t_{med} = 1min$, average latency values of both approaches keep the same behavior and close values for both static and dynamic scenarios.

For the results presented in Figure 5.11(b) with $N_s = 1000$, churn rate increases beyond usual values, i.e., it varies from 1% up 10% of the subscribers per minute. In this case, t_{med} varies from $69s$ to $7s$. Note that for the experiments shown in Figure 5.11(a) with $N_s = 1000$ and $t_{med} = 1min$, the churn rate is approximately 1.1% of the subscribers per minute. Although the higher the churn rate, the greater the number of messages over the network, it is possible to

observe that, in Figure 5.11(b), even if latency increases, *VCube-PS* tolerates well the increase in the number of messages: when the churn rate increases 10 times, latency grows in average 2.55 times, false positives represent in average 2.2% ($\sigma = 0.15\%$) of the `PUB` messages, and, in average, messages wait in queue no more than $28.36ms$ ($\sigma = 0.66ms$) before being forwarded. On the other hand, when the churn rate increases, *SRPT*'s single tree is not able to treat and send all the messages in time in order to avoid contention. In *SRPT*, with churn rate of 4% per minute ($t_{med} = 17s$) and 1000 subscribers, the overall number of sent messages is slightly smaller than that of the scenario with 4000 subscribers and $t_{med} = 1min$ (Figure 5.11(a)). In both cases, this is the point where *SRPT*'s reception latency starts to suffer from contention. Beyond this point, *SRPT*'s single root is unable to treat and forward messages without queuing them for long periods. For 5% churn rate per minute, messages are kept in queue, in average, $468ms$ ($\sigma = 185ms$) while for 10% churn rate up to $10s$ ($\sigma = 451ms$).

5.6.7 Broker-based SRPT

For the results presented in this section, it is also considered a *SRPT* Pub/Sub system based on brokers (e.g., DYNATOPS Zhao et al. (2013), see Section 3.5.1.1). Here, the broker-based approach is called *SRPT*-B and the previous subscriber-based one is renamed to *SRPT*-S. In *SRPT*-B, the single broadcast tree per topic is composed by nodes that are either brokers (instead of subscribers) or relays. Subscribers are directly connected to brokers, according to their locality and/or interests. Each published message for this topic is transmitted over this tree and each broker, upon reception, directly sends the message to the subscribers connected to it.

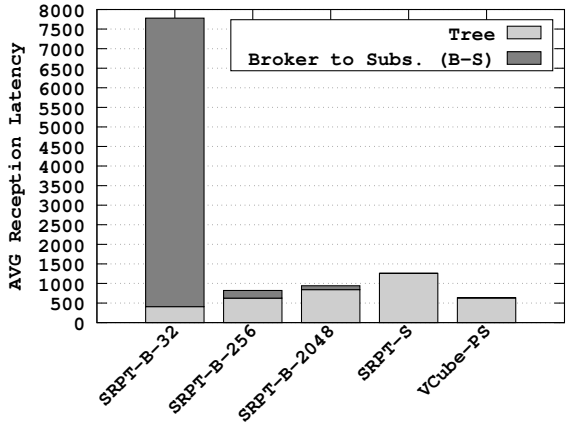


Figure 5.12: Average reception latency for different approaches and 4096 nodes.

Figure 5.12 shows the average reception latency for *SRPT*-S, *SRPT*-B, and *VCube-PS*. Publishers are randomly chosen among the subscribers of the topic and send a new message on average every 500 *u.t.*, up to a limit of 128 messages.

Three configurations for *SRPT*-B with different number of brokers were defined: 32, 256, and 2048. The other nodes are subscribers evenly distributed among the brokers: 127, 15, and 1 subscribers per broker, respectively. Note that, in this experiment, for both *SRPT*-B and *SRPT*-S, there is no relay node, i.e., trees are composed only by the respective numbers of brokers or by 4096 subscribers, respectively.

In *SRPT*-B, average reception latency is composed by the time to send the message to the brokers (`Tree` in the figure) plus the time for the broker to send the message to the connected subscribers (`B-S` in the figure). On the one hand, it is possible to in the figure that the fewer the number of brokers, the lower the `Tree` reception latency. On the other hand, the fewer the number of brokers, the higher the number of messages per broker forwarded to the subscribers, inducing broker-level contention (like the root in *SRPT*) and, therefore, the higher the `B-S` reception latency. In the configuration with 32 brokers, it is clear to see the high broker-level contention while, with 256 nodes, the load is better distributed.

It is also important to point out that even if *VCube-PS* builds trees with bigger height compared to *SRPT*-B's, it presents lower average reception latency than all *SRPT*-B configurations (22% better for *SRPT*-B with 256 brokers) since it avoids contention by exploiting multiple paths. A last observation is that *SRPT*-B with 2048 brokers has lower reception latency than *SRPT*-S since the latter presents more contention in the root of the tree, which is composed by 4096 subscribers.

5.7 Conclusion

Several existing topic-based approaches in the literature use trees to disseminate messages, where for each topic there exists a single broadcast tree. However, in case of high concentration of publications at the same topic (a "hot topic"), the root of the tree's topic can become a bottleneck. While several tree-based approaches use a single tree per topic, where a node is chosen as its root (*rendezvous* point), the proposed system builds, on top of a virtual hypercube-like topology, a distributed spanning tree rooted on the source of the broadcast node. Such trees are used to both propagate information about membership changes and disseminate published messages to subscribers. Furthermore, there is no permanent relay node, but, due to subscription dynamics, there may exist temporary ones.

Causal delivery order for messages published to the same topic is also ensured by the proposed system by using causal barriers. The latter is a structure that, instead of carrying per node information, carries only the identifiers of direct causally preceding messages. Besides the possibility of carrying less information when compared to vector-based approaches, causal barriers are also suitable for the subscription dynamics present in Pub/Sub systems because its implementation does not require knowledge about the number of participating nodes.

Simulations were implemented on top of *PeerSim*, comparing *VCube-PS* and single rooted approaches. Experimental results confirm that *VCube-PS* performs better when there is a high publication rate per topic, since it provides publication load balancing. Moreover, the decentralized broadcast of messages helps the causal dependencies to be delivered in a reasonable time. Finally, even if some messages can be delivered to nodes that are no longer subscribed to the message's topic, this is a temporary situation that affects only a small percentage of the overall number of messages.

Chapter 6

Conclusion

Contents

The notion of temporal order is an important concept in many computer systems. However, considering distributed systems where global clock is not available, the task of detecting such an order relies on a relation of cause and effect between events of a distributed computation known as the "happened-before" relation. Thereby, several works have used the concept of causality to devise logical time structures that keep track of the order of events.

Likewise, many distributed applications require group communication services that enable their processes to exchange messages respecting the "happened-before" relation of messages. One particularly important type of message ordering is the causal order, which is crucial to different types of applications, such as messaging services and distributed databases.

Besides the importance of offering services that respect causal order of messages, there exists a constant need for communication-efficient broadcast communication protocols. Several works have proposed to organize the nodes of the system in logical trees, in order to exploit the well-known logarithmic properties of the latter. However, these approaches use a single broadcast tree, which can induce root bottleneck due to message contention with, therefore, performance degradation.

A second issue related to broadcast protocols is the number of messages sent over the network. Studies have shown that the number of messages has a more significant impact on network performance than their sizes. Such a problem has been tackled by several existing solutions of the literature that, at the cost of increasing end-to-end message latencies, use timers to buffer several messages and then send them into a single one.

This thesis investigated how to provide a communication-efficient Publish/Subscribe (Pub/Sub) system. Particularly, this thesis focused on the topic-based model of Pub/Sub systems, through

which messages are classified according to known topics (e.g., keywords), and some topics present higher request rate than others ("hot topics").

6.1 Contributions

Both contributions of this thesis were built on top of a hypercube-like topology called *VCube* which presents strong logarithmic properties. By using such a topology, broadcast trees are dynamically built, rooted on the source of every broadcast message.

In Chapter 4, a new causal broadcast protocol was proposed, which reduces the traffic of messages through the network by aggregating messages without the use of timers. To this end, the algorithm considers variations in latencies of network links, causal relation between messages, and intersections that might exist in broadcast trees to decide whether messages can be aggregated or not. The idea is that the forwarding of a message can be delayed in order to be aggregated with another message, only if both messages will be forwarded by the node to the same child (children) and the first message causally depends on the second one. Such an aggregation mechanism does not induce any overhead since the sending of a message to a child node is worthless if this node will not be able to deliver it upon reception. In other words, the only messages that can be aggregated are those whose delivery latency would not be increased by the aggregation approach.

The causal aggregation broadcast protocol uses vector clocks to ensure causal order. Although other approaches presented in Section 3.3.3 (e.g. causal barriers) are more scalable than vector clocks, the latter enables the exploitation of the transitive property of causal order to deduce possible message aggregations.

The proposed causal aggregation broadcast protocol was implemented on top of the event-driven simulator *PeerSim*, and results showed that the bundling of several messages into a single one reduces message traffic as well as average delivery latencies since there is less node contention. Moreover, when receiving a packet with more than one message, a node is more likely to immediately deliver them to the application, reducing, therefore, the number of non deliverable pending messages.

The second contribution of the thesis, presented in Chapter 5, is a topic-based Pub/Sub system, *VCube-PS* which, unlike most existing solutions, also ensures per-topic causal broadcast of messages. It also addresses the problem of contention due to high concentration of messages transmitted through a same tree in Pub/Sub systems, in scenarios where most of the messages are related to few topics ("hot topics").

While most other tree-based Pub/Sub systems use a single tree and *rendezvous* points, *VCube-PS* dynamically creates a new spanning tree rooted on the source of every message that is published. By extending the tree construction algorithm (Chapter 2.4.1), the spanning tree built to disseminate publications associated to a given topic is composed only by current subscribers of the topic, thus reducing message traffic and latencies. Furthermore, this same tree structure is used to propagate subscription membership changes.

In order to track causal order of published messages, contrarily to the other contribution of

this thesis, *VCube-PS* uses causal barriers which are suitable to tackle subscription dynamics.

Results from experiments on top of *PeerSim* confirmed that, when compared to approaches with one single tree per topic, *VCube-PS* presents the lowest latency results under a high publication rate per topic, since it intrinsically provides load balancing. Moreover, *VCube-PS* generates less message traffic and the extra delay necessary to ensure causal delivery order represents only a small percentage of the end-to-end average message latencies. Finally, the presence of temporarily relay nodes increases reception latency of only a small number of publications.

6.2 Perspectives

This section presents a guideline for future directions of this thesis. The evolution of this work involves some activities that can be performed in a near future and others that demand further time and development.

6.2.1 Short Term

The first task will be to provide a ***proof of correctness*** of the protocols of both contributions.

A second task will be to apply the causal aggregation approach to the causal broadcast protocol of *VCube-PS*, which uses causal barriers to keep track of causal dependencies. On the one hand, causal barriers are suitable for coping with subscription dynamics present in Pub/Sub systems, because it is not dependent of the number of nodes in the system. On the other hand, vector clocks store significantly more information about transitive causality of messages than causal barriers. Therefore, the ***aggregation approach in VCube-PS*** could use either causal barriers, but limited to direct dependencies, or vector clocks, at the expense of scalability.

Finally, in the experimental results presented in Chapters 4 and 5, no assumption is made about the ***mapping between physical and logical nodes***. It would be interesting to provide strategies to map nodes according to, for instance, a latency matrix or communication locality.

6.2.2 Long Term

This thesis has been developed in the context of a French-Brazilian CNRS-INRIA-Fundação Araucária project entitled "Autonomic and Scalable Algorithms for Building Resilient Distributed Systems" whose members aim at keeping the cooperation, even if the project is over. Thus, the following long term activities are proposed.

6.2.2.1 Node Failure

Chapters 4 and 5 assume a system model where nodes do not fail and channels are reliable. Therefore, even if *VCube* is a distributed diagnosis algorithm, the contributions presented in this thesis exploited only its topology organization.

VCube can also be used as an underlying failure detector. The algorithm presented in Section 2.4.1 considers only correct nodes to construct spanning trees. However, upon detection

of failures during the dissemination of a message, it would be necessary to repair the tree and re-transmit messages.

Note that in *VCube-PS* (Chapter 5), it is possible to draw the analogy that when a node unsubscribes from a given topic, it can be seen as "faulty" in relation to that topic. Therefore, during the construction of a topic's spanning tree, only the "correct" nodes (current subscribers) of that topic are considered. However, in the proposed solution, when a node unsubscribes from a topic, it temporarily continues to participate in the topic's tree as a relay node. If node crashes are taken into account, a faulty node is no longer member of the topic and cannot act as a relay of messages, i.e., the tree must be reorganized and lost messages must be re-transmitted.

The causal aggregation broadcast protocol proposed in Chapter 4 deduces which received messages can be aggregated by taking into account common children. In case of failure, upon detection of it, broadcast trees should be reorganized and, consequently, path intersections change.

It is also worth pointing out that *VCube* considers a complete graph, detecting $N-1$ failures, being N the number of nodes. For scalability sake, the ideal would be that a correct node communicates only with its correct neighbors in the logical hypercube. On the other hand, in this case, the maximum number of tolerated failures is reduced.

6.2.2.2 Causal Aggregation and Timer-based Aggregation

The timer-based reliable broadcast protocol presented by Rodrigues et al. (2018) builds spanning trees as the one proposed in Chapter 4 but, unlike it, aggregates messages using timers. The two protocols could be modified to provide both reliability and causal order of broadcast messages.

Another work would be a new hybrid approach. For instance, if few messages get to be aggregated by using the causal approach because messages are not received out of causal order, timers could be used by some nodes (at the cost of increasing latency). A second possibility is to aggregate unrelated messages that are addressed to a same node (like Rodrigues et al. (2018)) but without using timers: if a node is waiting for some causal dependency of a message to be aggregated before sending them to a given child node, any other unrelated message that is also addressed to this child node could be aggregated during this waiting time.

6.2.2.3 Extension of *VCube-PS* to Geo-localization

In the proposed Pub/Sub system, topics are simply keywords. Another approach could consider topics as subdivisions of a geographical area, enabling geo-localized subscriptions. The area should be split into frames, each of them corresponding to a subscription unit, such as in R-Trees (Guttman, 1984). Hence, a subscription to an area consists of subscriptions to all the frames that compose (intersect) this area.

However, a new kind of false-positive appears: if a node subscribes to an area that is inside one (or several) frames, it would actually subscribe to the entire area comprising the involved frames. The Pub/Sub will need to apply filters to the published messages based on subscriber interest areas before delivering the messages to them. Hence, there exists a trade-off between subscription unit size (for coping with local filtering) and the cost of subscription management (in terms of amount of control messages and stored membership information).

Bibliography

Adelstein, F. and Singhal, M. (1995). Real-time causal message ordering in multimedia systems. In *Proceedings of 15th International Conference on Distributed Computing Systems*, pages 36–43.

Akkaya, K., Demirbas, M., and Aygun, R. S. (2008). The impact of data aggregation on the performance of wireless sensor networks. *Wirel. Commun. Mob. Comput.*, 8(2):171–193.

Arantes, L., Potop-Butucaru, M. G., Sens, P., and Valero, M. (2010). Enhanced dr-tree for low latency filtering in publish/subscribe systems. In *2010 24th IEEE International Conference on Advanced Information Networking and Applications*, pages 58–65.

Astley, M., Auerbach, J., Bhola, S., Buttner, G., Kaplan, M., Miller, K., Robert Saccone, J., Strom, R., Sturman, D. C., Ward, M. J., and Zhao, Y. (2004). Achieving scalability and throughput in a publish/subscribe system. Technical Report RC23103, IBM.

Bailis, P., Fekete, A., Ghodsi, A., Hellerstein, J. M., and Stoica, I. (2012). The potential dangers of causal consistency and an explicit solution. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 22:1–22:7, New York, NY, USA. ACM.

Baldoni, R., Beraldi, R., Quema, V., Querzoni, L., and Tucci-Piergiovanni, S. (2007). Tera: Topic-based event routing for peer-to-peer architectures. In *Proceedings of the 2007 Inaugural International Conference on Distributed Event-based Systems*, DEBS '07, pages 2–13, New York, NY, USA. ACM.

Baldoni, R., Bonomi, S., Platania, M., and Querzoni, L. (2012). Dynamic message ordering for topic-based publish/subscribe systems. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pages 909–920.

Baldoni, R., Querzoni, L., and Virgillito, A. (2005). Distributed event routing in publish/subscribe communication systems: a survey. Technical report.

Baldoni, R., Raynal, M., Prakash, R., and Singhal, M. (1996). Broadcast with time and causality constraints for multimedia applications. In *Proceedings of EUROMICRO 96. 22nd Euromicro Conference. Beyond 2000: Hardware and Software Design Strategies*, pages 617–624.

Bianchi, S., Felber, P., and Potop-Butucaru, M. G. (2010). Stabilizing distributed r-trees for peer-to-peer content routing. *IEEE Transactions on Parallel and Distributed Systems*, 21(8):1175–1187.

Birman, K., Schiper, A., and Stephenson, P. (1991). Lightweight causal and atomic group multicast. *ACM Trans. Comput. Syst.*, 9(3):272–314.

Birman, K. P. and Joseph, T. A. (1987). Reliable communication in the presence of failures. *ACM Trans. Comput. Syst.*, 5(1):47–76.

Blessing, S., Clebsch, S., and Drossopoulou, S. (2017). Tree topologies for causal message delivery. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, AGERE 2017, pages 1–10, New York, NY, USA. ACM.

Bloom, B. H. (1970). Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426.

Bravo, M., Rodrigues, L., and Van Roy, P. (2017). Saturn: A distributed metadata service for causal consistency. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 111–126, New York, NY, USA. ACM.

Cañas, C., Pacheco, E., Kemme, B., Kienzle, J., and Jacobsen, H.-A. (2015). Graps: A graph publish/subscribe middleware. In *Proceedings of the 16th Annual Middleware Conference*, Middleware '15, pages 1–12, New York, NY, USA. ACM.

Cai, W., Lee, B.-S., and Zhou, J. (2002). Causal order delivery in a multicast environment. *J. Parallel Distrib. Comput.*, 62(1):111–131.

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

Castro, M., Druschel, P., Kermarrec, A. M., and Rowstron, A. I. T. (2002). Scribe: a large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications*, 20(8):1489–1499.

Chang, E. J. H. (1982). Echo algorithms: Depth parallel operations on general graphs. *IEEE Transactions on Software Engineering*, SE-8(4):391–401.

Charron-Bost, B. (1991). Concerning the size of logical clocks in distributed systems. *Inf. Process. Lett.*, 39(1):11–16.

Chen, C., Tock, Y., and Girdzijauskas, S. (2018). Beaconvey: Co-design of overlay and routing for topic-based publish/subscribe on small-world networks. In *Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems*, DEBS '18, pages 64–75, New York, NY, USA. ACM.

Chen, C., Vitenberg, R., and Jacobsen, H.-A. (2016). Omen: Overlay mending for topic-based publish/subscribe systems under churn. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, DEBS '16, pages 105–116, New York, NY, USA. ACM.

Chetlur, M., Abu-Ghazaleh, N., Radhakrishnan, R., and Wilsey, P. A. (1998). Optimizing communication in time-warp simulators. In *Parallel and Distributed Simulation, 1998. PADS 98. Proceedings. Twelfth Workshop on*, pages 64–71.

Choi, Y., Park, K., and Park, D. (2004). Homed: a peer-to-peer overlay architecture for large-scale content-based publish/subscribe system. In *Proceedings of the third International Workshop on Distributed Event-Based Systems (DEBS)*, pages 20–25, Edinburgh, Scotland.

Crowcroft, J. and Paliwoda, K. (1988). A multicast transport protocol. In *Symposium Proceedings on Communications Architectures and Protocols*, SIGCOMM '88, pages 247–256, New York, NY, USA. ACM.

Cugola, G., Nitto, E. D., and Fuggetta, A. (2001). The jedi event-based infrastructure and its application to the development of the opss wfms. *IEEE Transactions on Software Engineering*, 27(9):827–850.

Cugola, G. and Picco, G. P. (2006). Reds: A reconfigurable dispatching system. In *Proceedings of the 6th International Workshop on Software Engineering and Middleware*, SEM '06, pages 9–16, New York, NY, USA. ACM.

de Araujo, J. P., Arantes, L., Duarte, E. P., Rodrigues, L. A., and Sens, P. (2017). A publish/subscribe system using causal broadcast over dynamically built spanning trees. In *Proceedings of the 29th International Symposium on Computer Architecture and High Performance Computing*, SBAC-PAD 2017, pages 161–168, Campinas, SP, Brazil.

de Araujo, J. P., Arantes, L., Duarte, E. P., Rodrigues, L. A., and Sens, P. (2018). A communication-efficient causal broadcast protocol. In *Proceedings of the 47th International Conference on Parallel Processing*, ICPP 2018, pages 74:1–74:10, Eugène, OR, USA.

de Araujo, J. P., Arantes, L., Duarte, E. P., Rodrigues, L. A., and Sens, P. (2019). Vcube-ps: A causal broadcast topic-based publish/subscribe system. *Journal of Parallel and Distributed Computing*, 125:18–30.

Défago, X., Schiper, A., and Urbán, P. (2004). Total order broadcast and multicast algorithms. *ACM Computing Surveys*, 36(4):372–421.

Duarte, E. P., Bona, L. C. E., and Ruoso, V. K. (2014). Vcube: A provably scalable distributed diagnosis algorithm. In *Proceedings of the 5th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, ScalA '14, pages 17–22, Piscataway, NJ, USA. IEEE Press.

Esposito, C., Cotroneo, D., and Russo, S. (2013). On reliability in publish/subscribe services. *Comput. Netw.*, 57(5):1318–1343.

Eugster, P. T., Felber, P. A., Guerraoui, R., and Kermarrec, A.-M. (2003). The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131.

Evropeytsev, G., Domínguez, E. L., Hernandez, S. E. P., Trinidad, M. A. L., and Cruz, J. R. P. (2017). An efficient causal group communication protocol for p2p hierarchical overlay networks. *Journal of Parallel and Distributed Computing*, 102:149 – 162.

Fidge, C. J. (1988). Timestamps in Message-Passing Systems that Preserve the Partial Ordering. In *11th Australian Computer Science Conference*, pages 55–66, University of Queensland, Australia.

Fidler, E., Jacobsen, H.-A., Li, G., and Mankovskii, S. (2010). The padres distributed publish/subscribe system. In *Principles and Applications of Distributed Event-Based Systems*, page 164–205.

Friedman, R. and Manor, S. (2004). Causal ordering in deterministic overlay networks. Technical report, Israel Institute of Technology, Haifa, Israel.

Gascon-Samson, J., Garcia, F., Kemme, B., and Kienzle, J. (2015). Dynamoth: A scalable pub/sub middleware for latency-constrained applications in the cloud. In *35th IEEE International Conference on Distributed Computing Systems, ICDCS 2015, Columbus, OH, USA, June 29 - July 2, 2015*, pages 486–496.

Girdzijauskas, S., Chockler, G., Vigfusson, Y., Tock, Y., and Melamed, R. (2010). Magnet: Practical subscription clustering for internet-scale publish/subscribe. In *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*, DEBS '10, pages 172–183, New York, NY, USA. ACM.

Girdzijauskas, S., Datta, A., and Aberer, K. (2007). Oscar: Small-world overlay for realistic key distributions. In Moro, G., Bergamaschi, S., Joseph, S., Morin, J.-H., and Ouksel, A. M., editors, *Databases, Information Systems, and Peer-to-Peer Computing*, pages 247–258, Berlin, Heidelberg. Springer Berlin Heidelberg.

Gupta, A., Liskov, B., and Rodrigues, R. (2004). Efficient routing for peer-to-peer overlays. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation - Volume 1*, NSDI'04, Berkeley, CA, USA. USENIX Association.

Guttman, A. (1984). R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, SIGMOD '84, pages 47–57, New York, NY, USA. ACM.

Hélary, J. and Mostefaoui, A. (1993). *A O(log2n) Fault-tolerant Distributed Mutual Exclusion Algorithm Based on Open-cube Structure.* Publications internes: Institut de Recherche en Informatique et Systèmes Aléatoires. sn.

Hidalgo, N., Arantes, L., Sens, P., and X. Bonnaire, X. (2010). An aggregation-based routing protocol for structured peer to peer overlay networks. In *AP2PS 2010 - 2nd International Conference on Advances in P2P Systems*, pages 76–81, Florence, Italy.

Kim, C. and Ahn, J. (2006). Epidemic-style causal order broadcasting only using partial view. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications & Conference on Real-Time Computing Systems and Applications, PDPTA 2006, Las Vegas, Nevada, USA, June 26-29, 2006, Volume 1*, pages 207–211.

Kim, K., Mehrotra, S., and Venkatasubramanian, N. (2010). Farecast: Fast, reliable application layer multicast for flash dissemination. In *Middleware*, volume 6452 of *Lecture Notes in Computer Science*, pages 169–190. Springer.

Kshemkalyani, A. D. and Singhal, M. (2008). *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge University Press, New York, NY, USA, 1 edition.

Kurose, J. F. and Ross, K. W. (2012). *Computer Networking: A Top-Down Approach (6th Edition)*. Pearson, 6th edition.

Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565.

Li, G. and Gao, S. (2011). Drscribe: An improved topic-based publish-subscribe system with dynamic routing. In *Proceedings of the 12th International Conference on Web-age Information Management*, WAIM'11, pages 226–237.

Li, M., Ye, F., Kim, M., Chen, H., and Lei, H. (2011). Bluedove - a scalable and elastic publish/subscribe service. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, pages 1254–1265, Washington, DC, USA. IEEE Computer Society.

Liebeherr, J. and Beam, T. K. (1999). Hypercast: A protocol for maintaining multicast group members in a logical hypercube topology. In Rizzo, L. and Fdida, S., editors, *Networked Group Communication*, pages 72–89, Berlin, Heidelberg. Springer Berlin Heidelberg.

Lumezanu, C., Baden, R., Spring, N., and Bhattacharjee, B. (2009). Triangle inequality variations in the internet. In *Internet Measurement Conference*, pages 177–183. ACM.

Lumezanu, C., Spring, N., and Bhattacharjee, B. (2006). Decentralized message ordering for publish/subscribe systems. In *Proceedings of the ACM/IFIP/USENIX 2006 International Conference on Middleware*, Middleware '06, pages 162–179, New York, NY, USA. Springer-Verlag New York, Inc.

Malekpour, A., Carzaniga, A., Carughi, G. T., and Pedone, F. (2011). Probabilistic fifo ordering in publish/subscribe networks. In *2011 IEEE 10th International Symposium on Network Computing and Applications*, pages 33–40.

Mattern, F. (1989). Virtual time and global states of distributed systems. In *Proceedings of the Workshop on Parallel and Distributed Algorithms*, pages 215–226. North-Holland.

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

Mostéfaoui, A. and Weiss, S. (2017). *A Probabilistic Causal Message Ordering Mechanism*, pages 315–326. Springer International Publishing, Cham.

Nakayama, H., Duolikun, D., Enokido, T., and Takizawa, M. (2016). Reduction of unnecessarily ordered event messages in peer-to-peer model of topic-based publish/subscribe systems. In *2016 IEEE 30th International Conference on Advanced Information Networking and Applications (AINA)*, pages 1160–1167.

Nédelec, B., Molli, P., and Mostéfaoui, A. (2018a). Breaking the scalability barrier of causal broadcast for large and dynamic systems. *CoRR*, abs/1805.05201.

Nédelec, B., Molli, P., and Mostefaoui, A. (2018b). Causal Broadcast: How to Forget? In *The 22nd International Conference on Principles of Distributed Systems (OPODIS)*, Hong Kong, China.

Patel, J. A., Rivière, E., Gupta, I., and Kermarrec, A.-M. (2009). Rappel: Exploiting interest and network locality to improve fairness in publish-subscribe systems. *Computer Networks*, 53(13):2304 – 2320. Gossiping in Distributed Systems.

Plesca, C., Grigoras, R., Queinnec, P., Padiou, G., and Fanchon, J. (2006). A coordination-level middleware for supporting flexible consistency in cscw. In *14th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP'06)*, pages 6 pp.–.

Postel, J. (1981). Internet protocol. STD 5, RFC Editor.

Prakash, R., Raynal, M., and Singhal, M. (1996). An efficient causal ordering algorithm for mobile computing environments. In *Proceedings of 16th International Conference on Distributed Computing Systems*, pages 744–751.

Ramaswamy, R., Weng, N., and Wolf, T. (2004). Characterizing network processing delay. In *Global Telecommunications Conference, 2004. GLOBECOM '04. IEEE*, volume 3, pages 1629–1634 Vol.3.

Raynal, M., Schiper, A., and Toueg, S. (1991). The causal ordering abstraction and a simple way to implement it. *Information Processing Letters*, 39(6):343 – 350.

Raynal, M. and Singhal, M. (1996). Logical time: capturing causality in distributed systems. *Computer*, 29(2):49–56.

Rhea, S., Geels, D., Roscoe, T., and Kubiatowicz, J. (2004). Handling churn in a dht. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '04, Berkeley, CA, USA. USENIX Association.

Rodrigues, L. A., Arantes, L., and Duarte, E. P. (2014). An autonomic implementation of reliable broadcast based on dynamic spanning trees. In *Dependable Computing Conference (EDCC), 2014 Tenth European*, pages 1–12.

Rodrigues, L. A., Duarte, E. P., de Araujo, J. P., Arantes, L., and Sens, P. (2018). Bundling messages to reduce the cost of tree-based broadcast algorithms. In *Proceedings of the 8th Latin-American Symposium on Dependable Computing*, LADC 2018, pages 115–124, Foz do Iguaçu, PR, Brazil.

Rowstron, A. I. T. and Druschel, P. (2001). Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, Middleware '01, pages 329–350, London, UK, UK. Springer-Verlag.

Sanli, C. and Lambiotte, R. (2015). Local variation of hashtag spike trains and popularity in twitter. *PLOS ONE*, 10(7):1–18.

Saroiu, S., Gummadi, K. P., and Gribble, S. D. (2002). A measurement study of peer-to-peer file sharing systems. In *Multimedia Computing and Networking (MMCN)*.

Schiper, A., Eggli, J., and Sandoz, A. (1989). A new algorithm to implement causal ordering. In *WDAG*, volume 392 of *Lecture Notes in Computer Science*, pages 219–232. Springer.

Schwarz, R. and Mattern, F. (1994). Detecting causal relationships in distributed computations: In search of the holy grail. *Distrib. Comput.*, 7(3):149–174.

Setty, V., van Steen, M., Vitenberg, R., and Voulgaris, S. (2012). *PolderCast: Fast, Robust, and Scalable Architecture for P2P Topic-Based Pub/Sub*, pages 271–291. Springer Berlin Heidelberg, Berlin, Heidelberg.

Shudo, K. (2017). Message bundling on structured overlays. In *2017 IEEE Symposium on Computers and Communications (ISCC)*, pages 424–431.

Sianati, A., Boukerche, A., and Grande, R. D. (2015). Bundling communication messages in large scale cloud environments. In *2015 IEEE Symposium on Computers and Communication (ISCC)*, volume 00, pages 788–795.

Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H. (2001). Chord: A scalable peer-to-peer lookup service for internet applications. *SIGCOMM Comput. Commun. Rev.*, 31(4):149–160.

Torres-Rojas, F. J. and Ahamad, M. (1999). Plausible clocks: constant size logical clocks for distributed systems. *Distributed Computing*, 12(4):179–195.

Tzeng, N. . and Wei, S. (1991). Enhanced hypercubes. *IEEE Transactions on Computers*, 40(3):284–294.

Vuillemin, J. (1978). A data structure for manipulating priority queues. *Commun. ACM*, 21(4):309–315.

Wang, G., Zhang, B., and Ng, T. S. E. (2007). Towards network triangle inequality violation aware distributed systems. In *Internet Measurement Conference*, pages 175–188. ACM.

Wang, J., Bahulkar, K., Ponomarev, D., and Abu-Ghazaleh, N. (2013). Can pdes scale in environments with heterogeneous delays? In *Proceedings of the 1st ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, SIGSIM PADS '13, pages 35–46, New York, NY, USA. ACM.

Wang, X., Zhang, Y., Zhang, W., Lin, X., and Wang, W. (2015). Ap-tree: efficiently support location-aware publish/subscribe. *The VLDB Journal*, 24(6):823–848.

Wang, Y., Fan, J., Jia, X., and Huang, H. (2012). An algorithm to construct independent spanning trees on parity cubes. *Theor. Comput. Sci.*, 465:61–72.

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

Yamamoto, Y. and Hayashibara, N. (2017). Merging topic groups of a publish/subscribe system in causal order. In *2017 31st International Conference on Advanced Information Networking and Applications Workshops (WAINA)*, pages 172–177.

Yang, J., Chang, J., Pai, K., and Chan, H. (2015). Parallel construction of independent spanning trees on enhanced hypercubes. *IEEE Transactions on Parallel and Distributed Systems*, 26(11):3090–3098.

Yestemirova, G. and Saginbekov, S. (2018). Efficient data aggregation in wireless sensor networks with multiple sinks. In *2018 IEEE 32nd International Conference on Advanced Information Networking and Applications (AINA)*, pages 115–119.

Zhang, K., Muthusamy, V., and Jacobsen, H. A. (2012). Total order in content-based publish/subscribe systems. In *2012 IEEE 32nd International Conference on Distributed Computing Systems*, pages 335–344.

Zhao, B. Y., Kubiatowicz, J. D., and Joseph, A. D. (2001). Tapestry: An infrastructure for fault-tolerant wide-area location and. Technical report, Berkeley, CA, USA.

Zhao, Y., Kim, K., and Venkatasubramanian, N. (2013). Dynatops: A dynamic topic-based publish/subscribe architecture. In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, DEBS '13, pages 75–86, New York, NY, USA. ACM.

Zhuang, S. Q., Zhao, B. Y., Joseph, A. D., Katz, R. H., and Kubiatowicz, J. D. (2001). Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination. In *Proc. 11th Int'l Work. Net. Oper. Systems Support for Digital Audio and Video*, NOSSDAV '01, pages 11–20, New York, NY, USA. ACM.