A Publish/Subscribe System Using Causal Broadcast Over Dynamically Built Spanning Trees

João Paulo de Araujo*, Luciana Arantes†, Elias P. Duarte Jr., Luiz A. Rodrigues§ and Pierre Sens¶

*† Sorbonne Universités, UPMC Univ Paris 06, CNRS, INRIA, LIP6 – Paris, France
† Floral University of Paraná – Curitiba, Brazil
§ Western Paraná State University – Cascavel, Brazil

Email: *joao.araujo@lip6.fr, † luciana.arantes@lip6.fr, ‡ elias@inf.ufpr.br, § luiz.rodrigues@unicente.br, ¶pierre.sens@lip6.fr

Abstract—In this paper we present VCube-PS, a topic-based Publish/Subscribe system built on the top of a virtual hypercube-like topology. Membership information and published messages to subscribers (members) of a topic group are broadcast over dynamically built spanning trees rooted at the message’s source. For a given topic, delivery of published messages respects causal order. Performance results of experiments conducted on the PeerSim simulator confirm the efficiency of VCube-PS in terms of scalability, latency, number, and size of messages when compared to a single rooted, not dynamically, tree built approach.

I. INTRODUCTION

Publish/Subscribe (Pub/Sub) systems consist of distributed nodes in which one or more publishers produce messages (events) that are consumed by subscribers. 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 [1], [2].

There basically exist two models of Pub/Sub systems: topic-based [3], [4], [5], [6] and content-based [7], [8]. In the first one, 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 then it receives all published messages related to these topics. In the content-based model, events are structured in multiple attributes, and subscribers express their interests by specifying constraints over the values of these attributes [9].

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 [1], 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. On the other hand, the content-based model provides more flexibility to subscribers for defining their event interests, but at the expense of more complex user interfaces and the need for filtering.

In this work, we are interested in topic-based Pub/Sub systems and, particularly, in offering an efficient support for publishers to send messages to subscribers, guaranteeing causal order among published messages, low latency, and load balancing.

In our proposed system, called VCube-PS, a user (node) can subscribe or unsubscribe to a topic. After becoming a subscriber of a topic and, therefore, member of the topic’s group, a node publishes a message associated to the topic. Message is sent to all subscribers of a topic using a broadcast protocol on top of a spanning tree, composed just by the subscribers, whose root is the publisher of the message. This tree is dynamically built on top of a virtual hypercube-like topology, called VCube [10], that presents several logarithmic properties. Moreover, the tree construction itself has no overhead, since it is built using information nodes already have. We consider that nodes do not fail but can dynamically join to (make new subscription) or leave from (cancel subscription) one or more topic groups.

VCube-PS also ensures causal ordering among published messages related to the same topic: if a node publishes a message after it delivers another message, then no node delivers the latter after the former. We note that in Pub/Sub systems the guarantee of published messages respecting causal order is an important and useful feature. For instance, in a discussion group, a question published on a group should never be delivered to any subscriber after an answer published in the same group, since question and answer are causally related. For sake of scalability, VCube-PS uses the causal barrier principle for implementing the causal order [11] where a message carries information about only those messages on which it directly depends.

Many Pub/Sub systems in the literature are based on broadcast trees [3], [4], [12]. However, they usually employ only one single generated tree, whose maintenance is sometimes costly when the membership of the system changes. Nodes which are not subscribers may take part in the tree as forwarders such as in Scribe Pub/Sub system [3], increasing the latency of message delivery, and the broadcast of a new publication is carried out by a single root node which, consequently, can become a bottleneck. In VCube-PS there is no single root node, as each node that publishes a message becomes the root of the corresponding spanning tree, providing load balancing.
balancing. Furthermore, a spanning tree is composed only by the subscribers of the same topic and does not include forwarder nodes. A third point to emphasize is that very few Pub/Sub systems in the literature (to the best of our knowledge, just JEDI [7]) provide causal ordering of published messages.

Experiments were conducted on the PeerSim simulator [13] and comparative results confirm the advantages of using per-publisher dynamically built spanning trees for load balancing, latency, number and size of messages, as well as the causal barrier approach for implementing causal ordering.

The rest of the paper is organized as follows. Section II introduces VCube. Section III presents the extensions and functions added to VCube in order to manage topics, order messages, and gives a description of VCube-PS’s algorithms. Section IV presents evaluation results obtained from experiments conducted on PeerSim simulator. Section V discusses related work and, finally, Section VI concludes the paper.

II. VCUBE

In VCube [10], a node i (also called p_i) groups the other n−1 nodes in d = log_2 n clusters forming a d-VCube, each cluster s (s > 0) has size 2^{s−1}. The ordered list of nodes in each cluster s is denoted by function c_{i,s} below, where ⊕ is the bitwise exclusive or operator (xor).

\[ c_{i,s} = i \oplus 2^{s-1} \| c_{i,0^{s−1}}, k \| k = 1, ..., s − 1 \]

VCube is a distributed fault diagnosis system and it defines as the neighbors of a node i the first faulty-free node of each cluster s. Periodically, i tests the first node in the c_{i,s} to check whether it is correct or faulty. Fig. 1 shows node p_0’s hierarchical cluster-based logical organization of n = 8 nodes 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 node p_0 and that there are no failures. The clusters of p_0 are shown in the same figure. 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 their respective cluster.

III. VCUBE-PS: PUBLISH/SUBSCRIBE SYSTEM

We consider a distributed system composed of a finite set of \( \Pi = \{p_1, ..., p_{n-1}\} \) nodes (users) with \( n = 2^d \) processes, \( d > 0 \). Each node has a unique id and nodes communicate only by message passing. Each single node executes one task and a user of the Pub/Sub system corresponds to a node. Therefore, the terms node, user, and process are interchangeable in this work.

Nodes communicate by sending and receiving messages. The network is fully connected: each pair of nodes is connected by a bidirectional point-to-point channel and there is no network partitioning. Nodes do not fail and links are reliable. Thus, messages exchanged between any two processes are never lost, corrupted or duplicated. Each message is uniquely identified by the source id (s), the topic (t), and sequence number (c) given by the logical clock of the source. The system is asynchronous, i.e., relative processor speeds and message transmission delays are unbounded.

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, as we consider that 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 node of each cluster s in c_{i,s} in relation to topic t should also be a subscriber of t.

The following functions are provided to node i of VCube-PS by the virtual topology:

- \( \text{FF}_\text{NEIGHBOR}_i(t, s) \): if \( t = \ast \), no topic is considered and the function returns the first node in c_{i,s}; otherwise it returns the first node in c_{i,s} which is also a member of group t.
- \( \text{NEIGHBORHOOD}_i(t, h) \): returns a set that contains all nodes virtually connected to i according to FF\_NEIGHBOR\_i(t, s), for \( s = 1, ..., h \) and topic t. The parameter h can range from 1 to log_2 n. For \( h = \log_2 n \) the function returns all neighbors of i, if \( t = \ast \), no matter the topic; otherwise all neighbors of i that are subscribers of t. For any other value of \( h < \log_2 n \), the function returns only a subset of the first neighbors, i.e., those first neighbors whose respective cluster number s is smaller or equal to h (s ≤ h). For instance, in Fig. 1, if \( t = \ast \), \( \text{NEIGHBORHOOD}_0(\ast, 3) = \{1, 2, 4\} \), \( \text{NEIGHBORHOOD}_0(\ast, 2) = \{1, 2\} \), and \( \text{NEIGHBORHOOD}_4(\ast, 2) = \{5, 6\} \). On the other hand, if only nodes 0, 3, and 4 have joined topic t_1, \( \text{NEIGHBORHOOD}_0(t_1, 3) = \{3, 4\} \) and \( \text{NEIGHBORHOOD}_4(t_1, 2) = \ast \).
- \( \text{CLUSTER}_i(j) \): The function returns the index s of the cluster of node i that contains node j, (1 ≤ s ≤ \log_2 n). For instance, in Fig. 1, \( \text{CLUSTER}_0(1) = 1 \), \( \text{CLUSTER}_0(2) = \text{CLUSTER}_0(3) = 2 \), and \( \text{CLUSTER}_0(4) = \text{CLUSTER}_0(5) = \text{CLUSTER}_0(6) = \text{CLUSTER}_0(7) = 3 \).

**Causal and Per-source FIFO Reception Orders**: For each topic, VCube-PS respects causal order of published messages,
implementing, thus, a causal broadcast.

Causal Order: if a process publishes a message $m'$ after it has delivered another message $m$, then no process in the system will deliver $m$ after $m'$.

Note that if a process $i$ never delivers $m'$ (e.g., $i$ leaves the system before delivering $m'$) or delivers $m'$ but never delivers $m$ (e.g., $i$ was not in the system when $m$ was published), the causal order of published messages is not violated.

In order to implement the causal order of published messages, we apply causal barriers [11]. The advantage of the causal barrier approach is that it does not control causality based on nodes’ identifiers (per node vector) but by using direct dependencies of messages which renders the algorithm more scalable and suitable for dealing with the dynamics of nodes (subscriptions and unsubscriptions).

Let $m$ and $m'$ be two application messages published for topic $t$. Message $m$ immediately precedes message $m'$ (denoted $m \prec_{im} m'$) if (1) the publishing of $m$ causally precedes the publishing of $m'$ and (2) there exists no message $m''$ such that the publishing of $m$ causally precedes the publishing of $m''$, and the publishing of $m''$ causally precedes the publishing of $m'$. The causal barrier of $m$ ($cb_m$) consists of the set of messages that are immediate predecessors of $m$.

Fig. 2 shows 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 side of the figure, we have a timing diagram with the publishing and delivering of messages and on the right the graph with message dependencies. We can observe 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 $m_{1,t,2}$, not included, therefore, in $cb_{m_{1,t,2}}$.

![Fig. 2. Example of causal barrier.](image)

Let’s now suppose, in the same figure, that $p_3$ subscribes to $t$ after message $m_{2,t,1}$ was published to the other nodes, i.e., in this case, node $p_3$ will never receive nor deliver $m_{2,t,1}$. Hence, after having delivered $m_{1,t,1}$, $p_3$ can deliver $m_{1,t,2}$.

Since nodes can dynamically subscribe to or unsubscribe from a topic in VCube-PS, our implementation of message 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 the following property on the FIFO order of messages published on a given topic.

Per-source FIFO Reception Order: messages published by a same publisher are received by subscribers in the same order as they were produced. In other words, $m_{s,t,c}$ can be broadcast to the current subscribers of $t$ provided that the previous message $m'_{s,t,c-1}$ was received (not necessarily delivered) by all the subscribers present in the system when $m'$ was broadcast. This order allows a subscriber of $t$ to know that it will never receive any messages previously published, i.e., if $m'_{s,t,c'}$ is the first message that $i$ receives from $s$ on topic $t$ after it joined $t$’s group, $i$ will never receive $m_{s,t,c}$, if $c < c'$.

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 had previously 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.

VCube-PS Algorithm Description: Due to lack of space, we do not present the algorithms that manage topic membership and disseminate published messages to subscribers of a given topic. The latter can be found in Section III of [14]. In the following, we give a description of the algorithms.

Application Functions: VCube-PS offers to the application the functions $\text{SUBSCRIBE}(t)$, $\text{UNSUBSCRIBE}(t)$, and $\text{PUBLISH}(t, m)$ which allow a user (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 provided it is currently a subscriber of this topic. Those functions will generate messages of type $\text{SUB}$, $\text{UNS}$, and $\text{PUB}$ respectively which will be sent to all nodes, in case of subscription, or to all subscribers of $t$, otherwise. Every message is uniquely identified by the source id, the topic, and a sequence number (counter) in regard to this topic. In the case of a message of type $\text{PUB}$, it also carries the identifiers of direct causally related messages (causal barrier). After a function is invoked, a distributed spanning tree dynamically built and rooted at the caller is used to disseminate the corresponding messages.

Producer/consumer approach: Since in Pub/sub systems the publishing of a message must be decoupled from its delivery by the subscribers, in VCube-PS, every message generated by the execution of a function by $i$ is inserted in a queue per topic $t$. Then, a per topic task at $i$ continuously removes the first message $m$ from this queue and starts its broadcast. The next message is removed from the queue only when $i$ is sure of the reception of $m$ by all current subscribers (per-source FIFO reception order), i.e., reception of acknowledgement (message of type $\text{ACK}$) from those nodes to whom $i$ sent $m$.

Propagation of a message: For every message $m$ taken from the queue of node $i$ associated to topic $t$, the corresponding task starts the propagation of $m$ by dynamically building a
hierarchical spanning tree, rooted at \( i \), composed by the nodes which are either subscribers of \( t \), in case of messages of type UNS or PUB, or composed by all nodes, in case of messages of type SUB. For this purpose, \( i \) calls function \( \text{NEIGHBORHOOD}(t, h) \) which renders the set of the first subscribers of \( t \) for each of the \( \log_2 n \) clusters of process \( i \) (in the case of \( \text{SUB} \) messages, \( t = \ast \ast \)). These neighbors become \( i \)'s children in the spanning tree and \( m \) is sent to them. Upon the reception of \( m \), by calling function \( \text{CLUSTER}_j(i) \), every \( i \)'s child \( j \) sends \( m \) to its \( s-1 \) first neighbors of \( j \) in relation to topic \( t \) and the cluster \( s \) of \( i \) to which \( j \) belongs, i.e., \( c_{i,s} \). These neighbors then become \( j \)'s children, and the process continues as described in the following example.

For instance, consider in Fig. 1 that either node \( p_0 \) is not a subscriber to \( t_1 \) and wants to subscribe or it is already a subscriber and wants to publish a message related to \( t_1 \). All other nodes are already subscribers. Node \( p_0 \) sends a message \( m \) related to \( t_1 \) (\( \text{SUB} \) or \( \text{PUB} \)), becoming then the root of the respective spanning tree. The message will be sent to the \( \log_2 n = 3 \) children of \( p_0 \): \( \text{FF}_\text{NEIGHBOR}_0(t_1, 1) \) = 1, \( \text{FF}_\text{NEIGHBOR}_0(t_1, 2) \) = 2, and \( \text{FF}_\text{NEIGHBOR}_0(t_1, 3) \) = 4. Upon the reception of \( m \), \( p_1 \) does not forward it since its \( \text{CLUSTER}_1(0) \) = 1, while \( p_2 \) (\( \text{CLUSTER}_2(0) \) = 2) forwards it to its child \( p_3 \), the first subscriber of its cluster \( 1 \) (\( c_{2,1} \)). When \( p_3 \) receives \( m \), as its \( \text{CLUSTER}_3(2) \) = 1, \( p_3 \) does not forward \( m \) to any node. However, in the case of \( p_4 \) (\( \text{CLUSTER}_4(0) = 3 \)), it forwards \( m \) to its children \( p_5 \in c_{1,4} \) and \( p_6 \in c_{4,2} \). Finally, since \( \text{CLUSTER}_6(4) = 2 \), \( p_6 \) sends \( m \) to \( p_7 \).

Let's consider now a second example also using the same Fig. 1 where only \( p_0, p_2, p_4, p_5 \), and \( p_6 \) nodes are subscribers of \( t_2 \) and \( p_2 \) publishes \( m' \) related to \( t_2 \). In this case, \( p_2 \) sends \( m' \) to the first neighbors of its \( \log_2 n = 3 \) clusters that are also subscribers of \( t_2 \): \( \text{FF}_\text{NEIGHBOR}_2(t_2, 1) \) = \( \perp \) (there is no subscriber \( t_2 \) in cluster \( c_{2,1} \)), \( \text{FF}_\text{NEIGHBOR}_2(t_2, 2) \) = 0, and \( \text{FF}_\text{NEIGHBOR}_2(t_2, 3) \) = 6 (\( p_6 \) is the first subscriber of \( t_2 \) in \( c_{2,3} \)). Upon receiving \( m' \), \( p_6 \) does not forward it to any node since the latter is not a subscriber of \( t_2 \). On the other hand, \( p_6 \), in the cluster \( c_{6,3} \) = \( (4, 5) \), \( p_4 \) is also a subscriber of \( t_2 \), and therefore sends \( m' \) to \( p_4 \) which then sends \( m' \) to \( p_5 \), the first and only \( t_2 \)'s subscriber of cluster \( c_{4,1} \).

After forwarding a message \( m \) to a child \( j \), node \( i \) waits for an ACK message from \( j \), which confirms the receipt of \( m \) by \( j \). However, \( j \) will only send an ACK back to \( i \) after receiving itself ACK messages from all its current children. ACK messages will, thus, be propagated from the leaves to the root, the source node of \( m \). Eventually, the latter receives all the ACK messages from its children and, in this case, the task related to \( t \) removes the next message to be published from the queue of the topic \( t \), if one exists. These sequences of \( \text{SUB} \), UNS, or PUB and then ACK messages from/to the source ensure the per-source FIFO reception order of published messages of a topic. In the case of \( \text{SUB} \) messages, the ACK messages will gather information about the membership of \( t \) and, therefore, \( i \) becomes aware of current subscribers.

**Reception and delivery of messages:** When receiving a PUB message \( m \) of topic \( t \) from \( s \), if \( i \) is a subscriber of \( t \) and has not already delivered \( m \), \( i \) keeps \( m \) and then verifies, based on direct causal dependencies, which received messages can be delivered to the application. A message \( m \) can be delivered to \( i \) only when every message \( m' \) on which \( m \) causally depends has been either already delivered by \( i \) or will never be received by \( i \) since \( \text{VCube-PS} \) has not considered \( i \) as a subscriber of \( t \) during the construction of the spanning tree which broadcast \( m' \). This determination of the first message is possible thanks to the fact that, for the same source, the publication of messages of the same topic respect per-source FIFO order. In the case of a \( \text{SUB} \) (resp., UNS) message, \( i \) includes (resp., removes) the identity of the source of the message in (resp., from) its \( t \) group membership.

**IV. Experimental Results**

In order to assess the performance of \( \text{VCube-PS} \) with different configuration scenarios, we conducted experiments on top of the event-driven PeerSim [13] simulator. In the majority of scenarios, we compared \( \text{VCube-PS} \) with an approach denoted \( \text{SRPT} \) \((\text{Single Root Per Topic})\), which is similar to Scribe [3], where a single multicast tree per topic, composed by both subscribers and non-subscribers (forwarders) is used to publish all messages of a topic. \( \text{SRPT} \) maps each topic into a node that acts as the root of the broadcast tree for the respective topic. Forwarders receive and re-transmit published messages but do not deliver them.

In the experiments, we consider that each message exchanged between two processes consumes \( t_{pc} + t_q + t_t + t_{pp} + t_d \) units of time (u.t.). Apart from \( t_d \) which represents the time necessary for a subscriber to satisfy all causal dependencies before delivering a message to the application, all other components are based on a packet-switched network delay model [15]: \( t_{pc} \) accounts for the processing time of a message by a node, e.g., checksum verification and routing decisions; \( t_q \) is the time a message must wait in the queue before being transmitted; \( t_t \) is the time necessary to transmit all bits of the message into the link, and \( t_{pp} \) expresses how long it takes for a message to transverse the link and reach the next hop. Assuming that there is no broadcast facility available in the system, if a message is sent to multiple destinations, a copy of it is queued for each of the destinations. For our experiments, the ratio between \( t_{pc} \) and \( t_{pp} \) has an impact on the threshold value for starting to queue messages as well as how fast the queue grows. Hence, based on [16], we set \( t_{pc} = t_t = 1 \) u.t. and \( t_{pp} = 100 \) u.t. \((1/100 \text{ ratio})\). The time a message stays queued \( (t_q) \) is a function of the rate of incoming/outgoing messages and can vary for each message. Likewise, when a node \( i \) receives a message \( m \), \( t_d \) will depend on how fast \( i \) will receive all the preceding messages of \( m \). If there is no preceding message \( t_d = 0 \).

The number of nodes \( n \) used in the simulations varies from 8 up to 1024, in a power of two. Each experiment was executed 40 times and for the results that comprise average values we also provide the standard deviation.

We consider the following metrics:
- **Latency**: the time that a published message takes to be 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 forwarders of messages of type PUB.

**Scenario with one publisher**: The aim of this experiment is to evaluate the impact of the logarithmic properties of **VCube-PS**. We consider that one publisher broadcasts only one message. Hence, when a subscriber receives the message, there is no delay for delivering it, since causal order of published messages is not applied in this case. Fig. 3 depicts the delivery latency when the number of nodes of the system varies and either 100% or 25% of them are subscribers. The set of subscribers is randomly chosen following a uniform distribution and fixed at the beginning of each execution. We should remark that when 100% of the nodes are subscribers, SRPT has no forwarder and, therefore, the latency for 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.

**Fig. 3.** Average latency for the delivery of one message to each subscriber.

In the case of 1024 nodes with 25% of subscribers uniformly distributed, the latency in **VCube-PS** is in average 420 units of time which results in a reduction of 31% compared to the latency presented by SRPT in the same scenario (620 u.t.). The higher standard deviation observed for SRPT with 25% of subscribers in scenarios with fewer nodes is due to the greater number of forwarders that have an impact in the average of the delivery time of all nodes. On the other hand, as the number of nodes increases, the number of samples (delivery time at each node) tends to a similar value as we average all executions. Moreover, with 100% of subscribers in SRPT, the standard deviation may be higher than zero since there exists a probability that the root node is the publisher itself and, therefore, SRPT performs just like **VCube-PS**, with no additional hop.

Forwarders induce an increase in the number of messages propagated throughout the network. Fig. 4 depicts the average number of PUB messages for the two approaches with 25% of the nodes as subscribers. In **VCube-PS**, this number is always equal to the number of subscribers, since there is no forwarder in the multicast tree. On the other hand, the forwarders in SRPT are responsible for up to 2.7 times more messages (for 8 nodes) when compared to **VCube-PS**. As the number of nodes increases, this difference is reduced, although **VCube-PS** generates, in average, at least 43% fewer messages than SRPT (1024 nodes).

**Scenario with several publishers**: In these experiments, 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 chosen by a uniform distribution between \([0, 1000]\) units of time. Fig. 5 shows in logarithmic scale the average reception latency when the number of nodes of the system varies and either 100% or 25% of them are subscribers. Since the ratio between the processing time \( (t_{pc}) \) and the propagation time \( (t_{pp}) \) has an impact on the load contention, we consider the ratio 1/100 (Fig. 5(a)), which is used in all other evaluation of this work, but also a propagation time which is ten times greater, \( (t_{pp} = 1000 \text{ u.t.}) \), leading to a ratio 1/1000 (Fig. 5(b)).

We can observe in Fig. 5(a) the stability of **VCube-PS** for both percentages of publishers, with a maximum increase of 5.7% (1024 nodes and 100% of publishers). 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 node of the tree at the same time they will be queued before transmission, increasing, thus, the reception latency. The numbers for SRPT up to 128 nodes are in average one hop in time higher compared to **VCube-PS** due to the additional hop used by SRPT. The arrival rate of messages in this case is close to the output rate, 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.

Comparing Fig. 5(b) with Fig. 5(a), we observe a lower increase in the average reception latency for SRPT in relation to **VCube-PS** since a 1/1000 ratio induces a higher latency in the reception of messages by a node, although its output throughput remains the same. Fig. 5(b) shows an increase of...
only 37% for 256 nodes (100% of publishers).

Table I shows the distribution of messages among nodes. For 1024 nodes and a ratio of 1/100 where all nodes are publishers and subscribers, the table groups nodes with similar average size of queue (given by intervals).

We observe an uneven distribution of the load among the nodes in SRPT when compared to VCube-PS: 98% of the nodes in VCube-PS have an average load between (4, 16] messages while, although 44% of the nodes in SRPT have in average between (0, 2] messages in their buffers, 50% of the nodes simply do not participate of the routing of any message. Moreover, in SRPT, one node (the root of the tree) has an average load of 9240 (σ = 4617) messages, which is a bottleneck that increases the overall reception latency and limits the load of the other nodes to just a few messages. Besides that, since each node publishes one message, in VCube-PS we have 1024 different trees which leads to an uniform distribution of sent messages where any node transmits exactly 1024 messages. On the other hand, in SRPT the relationship between the number of nodes and the amount of messages they publish is reduced exponentially.

Ordering of messages: In order to evaluate the size of messages and the latency due to message ordering, we consider 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. Additionally, a node $k$ has to wait at least $p$ messages before broadcasting its own. To this end, there are $j \geq 1$ 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.

Fig. 6 groups messages according to the size interval of their causal barriers for VCube-PS. When it is necessary to wait 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 all messages have just one causal dependency. On the other hand, if a node has to wait for more messages, 10 in the case of Fig. 6, 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, due to both the use of causal barriers and multiple tree-roots used by VCube-PS, the size of the dependency list of most of the messages keeps close to the dependency threshold.

It is also worth to evaluate the additional delay imposed by the causal barrier before delivering a message to the application. If messages take too long to be delivered, this delay may lead to a cascade effect that could make the system unfeasible. In our simulations, 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%).

**Scenario with several topics:** In order to evaluate the popularity of topics in Twitter, the authors in [17] carried out some experiments that show that roughly 60% of the topics have only one message published and 83% of them have no more than 5. On the other hand, 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 article.

![Fig. 7. Average reception latency with 256 nodes and 128 topics for two distribution of messages per topic.](image)

Table II: **Dynamics of subscriptions.**

<table>
<thead>
<tr>
<th>Churn</th>
<th>Nodes</th>
<th>AVG false-positives (std dev)</th>
<th>AVG deliveries (std dev)</th>
</tr>
</thead>
<tbody>
<tr>
<td>12.5%</td>
<td>512</td>
<td>2433.1 (764.4)</td>
<td>97445.5 (655.7)</td>
</tr>
<tr>
<td>12.5%</td>
<td>1024</td>
<td>10525.7 (1207.1)</td>
<td>194506.8 (943.6)</td>
</tr>
<tr>
<td>25%</td>
<td>512</td>
<td>3475.9 (899.9)</td>
<td>96193.9 (1001.7)</td>
</tr>
<tr>
<td>25%</td>
<td>1024</td>
<td>14590.4 (1717.8)</td>
<td>191760.3 (1549.8)</td>
</tr>
</tbody>
</table>

The number of false-positives reaches at most 7.6% of the total number of delivered messages (1024 nodes with 25% of churn). In this case, 192 nodes leave the topic at the same time. In the best case, all other subscribers are notified in up to $(\log_2 n + t_{pp})$, but it is necessary to consider the queuing of messages in other nodes of the spanning tree (including unsubscriptions and new subscriptions). With 1024 nodes and 75% of them as subscribers, 196608 messages should be delivered. However, when there is 25% of churn, in average 97.5% of these messages are delivered.

**V. RELATED WORK**

Numerous Pub/Sub systems in the literature, such as Scribe [3], Bayeux [4], DYNATOPS [6], and Dynamoth [5], are based on topics. Compared to content-based, topic-based systems provide simpler and more efficient implementations. They are usually deployed in contexts where efficient and fast notifications are required.

Similarly to VCube-PS, many Pub/Sub systems use tree-based overlays (e.g., Scribe [3], Bayeux [4], Marshmallow [12], DR-Tree [8], DYNATOPS [6]). The advantage of using trees is the logarithmic guarantees on publication delivery time and the number of messages employed. However, different from VCube-PS, most solutions often employ one single multicast tree (usually one per topic in topic-based systems), statically constructed from the start or as nodes join the system. Consequently, every publication should be
broadcast from the root of this tree that might, then, become a bottleneck. Moreover, many of these multicast trees include unrelated intermediate hops and nodes that are not subscribers which have to forward the message, presenting thus the problem of false positives and the need of message filtering (e.g. DR-Tree [8], Scribe [3]). Finally, their maintenance cost is usually high, specially in presence of churn.

Several solutions (e.g. Scribe [3], DYNATOPS [6], etc.) construct independent multicast trees on top of Distributed Hash Table (DHT) overlays (e.g Pastry, CAN). They adopt the rendezvous point approach, where a node, responsible for the hashed key of a topic name, becomes the rendezvous point, i.e., the root of the multicast related to the topic. Some DHT overlay like PeerCube [18] and HOMED [19] are based on a hypercube-like topology themselves.

Even if defining a coherent order for notifications in Pub/Sub systems is fundamental, few of them support event ordering [20], [7], [21], [22], specially total order. The authors in [20] propose a top-basic Pub/Sub system where messages published on different topics are either delivered in the same order to all subscribers or tagged as out-of-order (weak total order); while in [21], the task of ordering messages is distributed across sequencer nodes which totally order messages for the same topic. Considering FIFO links, Zhang et al. present in [22] a distributed total order protocol for a content-based Pub/Sub system where a broker can determine if a message can be delivered immediately or, by collaborating with other brokers, that a consistent delivery order is required. JEDI [7] is a Pub/Sub system that ensures causal order. The latter is implemented by the use of a return value, a message by which a receiver notifies an event delivery to the producer of the event, unlike VCube-PS, which does not require these extra messages since the causal dependencies of a message are included in the message itself (causal barriers).

VI. CONCLUSION

This work presented VCube-PS, a distributed topic-based Pub/Sub system. Spanning trees are dynamically built on the top of a virtual hypercube-like topology and are used to both propagate information about membership changes and disseminate published messages to subscribers. While other approaches use a node as the rendezvous point of the tree of a topic, we configure a distributed spanning tree rooted on the source of every message, without any extra cost due to VCube. The trees contain only subscribers of the topic which induce trees of shorter height when compared to a per-topic single root tree and, therefore, lower latencies and smaller number of messages. VCube-PS also provides causal ordering of messages whose implementation uses the causal barrier approach, adapted to cope with the dynamics of the system.

Experimental results on PeerSim confirm the logarithmic behavior of VCube-PS. Compared to an approach with one single root per topic, our solution performs better when there is a high publication rate per topic since it provides load balancing of publication. Furthermore, VCube-PS does not present permanent forwarders which induce false positives, but, due to subscription dynamics, only temporary ones which eventually do not take part in any spanning tree.

ACKNOWLEDGMENTS

This work was supported by a scholarship (PhD – GDE) from CNPq (Brazil) and by Fundação Araucária/SETI (Brazil) under the project 45112, grant 144/15.

REFERENCES