

SAPIENZA UNIVERSITÀ DI ROMA  
DIPARTIMENTO DI INFORMATICA E SISTEMISTICA  
ANTONIO RUBERTI

# Dynamic Message Ordering for Topic-Based Publish/Subscribe Systems

Roberto Baldoni, Silvia Bonomi, Marco Platania and Leonardo Querzoni  
[baldoni,bonomi,platania,querzoni]@dis.uniroma1.it

MIDLAB TECHNICAL REPORT 9/2011

## Abstract

A distributed event notification service (ENS) is a middleware architecture commonly used to provide applications with scalable and robust publish/subscribe communication primitives. A distributed ENS can route events toward subscribers using multiple paths with different lengths and latencies; as a consequence, subscribers can receive events out of order. In this paper, we propose a novel solution for out-of-order notification detection on top of an existing topic-based ENS. Our solution guarantees that events published on different topics will be either delivered in the same order to all the subscribers of those topics or tagged as *out-of-order*. The proposed algorithm is completely distributed and is able to scale with the system size while imposing a reasonable cost in terms of notification latency. Our solution improves the current state of the art solutions by dynamically handling subscriptions/unsubscriptions and by automatically adapting with respect to topic popularity changes.

## 1 Introduction

Modern large-scale applications are usually built on top of asynchronous communication primitives able to mask the unreliability of low-level networks and the dynamism of the application participants by decoupling in space and time the interacting parties. The publish/subscribe paradigm provides communication services where message addressing is implicitly handled by an *Event Notification Service* (ENS) that matches the content of events produced by publishers against interests expressed by subscribers in the form of subscriptions.

Many research efforts in publish/subscribe systems focussed on reliability and performance aspects with few contributions in the area of event ordering [15, 16, 17, 11]. Defining a coherent specification for notification ordering is a fundamental step for a wide range of applications like online games [4], stock tickers, messaging, or those based on composite event detection [19]. All these applications assign a semantic to the order in which events are notified; therefore, it is important that a notification ordering is specified and that the underlying ENS is able to guarantee its adherence to this specification or, at least, to provide hints about which subsets of notifications are guaranteed to be notified “in order”.

In this paper, we address the following simple ordering problem: how to guarantee that two subscribers sharing (at least) two same subscriptions are notified about events matching those subscriptions in the same order. While the above ordering problem stems from the simple rationale that two participants should always see the notification of two events in the same order, its enforcement in distributed ENSs is far from being trivial. Violations to the ordering property can easily arise due to the fact that two events, possibly published by different publishers, can follow distinct paths

through the ENS before reaching the point where they will be notified to the final recipients<sup>1</sup>. The impact of this problem can be easily seen by running a simple experiment executed on a toy application where two subscribers receive events published by two sources on two different topics, and check the occurrence of a specific notification pattern (e.g. the sequence of notifications  $e \rightarrow e' \rightarrow e''$ ); the results we obtained by running this test in a simple setting where events are diffused using SCRIBE [7] show how the ENS notifies events in a best-effort fashion without providing any form of ordering, thus allowing the two subscribers to coherently detect only about 35% of the patterns (further details on this test are reported in Section 6). Current approaches to solve this problem either (i) use hardware-based synchronization solutions to timestamp events [15] or (ii) implement total order [11] among all the receiving participants by trading performance for strong ordering guarantees, or (iii) give up some ordering aspects only guaranteeing per-source ordering [21] or, finally, (iv) require complex offline set-ups that must be continuously updated when subscribers change their interests [16].

In this paper we present a novel algorithm for out-of-order notification detection in distributed topic-based systems. Our solutions, encapsulated within a software component that can be deployed on top of any existing topic-based ENS, transparently delivers events notified by the ENS to the application layer and is able to deterministically tag every event whose notification violates the following *total notification order* property: if two independent subscribers are notified about the same two events, then these two events will be notified to them in the same order<sup>2</sup>. Out-of-order detection is realized by comparing logical timestamps that our algorithm automatically generates and attaches to events. The algorithm can use a configurable buffer to re-order events prior to notification as this can easily reduce the number out-of-order notifications. The algorithm performance have been analyzed through an extensive experimental study whose results show how our solution has a small impact on the event diffusion latency and the ability to dynamically adapt its behaviour to the current topic popularity distribution. Finally, we developed a prototype implementation of our algorithm, whose performance has been compared to a solution based on the JGroups [13] toolkit.

The rest of this paper is organized as follows: Section 2 introduces the system model and states the problem explaining why its solution includes some difficult aspects; Section 3 describes our algorithm and proves its correctness. In Section 4 we show how the algorithm respects also the causality relation among events published on the same topic; Section 5 presents some engineering aspects; Section 6 reports the results of the evaluation of

---

<sup>1</sup>Non-determinism in the form of unpredictable network latencies and message losses can easily exacerbate the problem.

<sup>2</sup>Note that this delivery order does not necessarily correspond to the real-time event production order.

our solution; Section 7 explains how the problem of ordering events in publish/subscribe systems has been tackled in the literature and, finally, Section 8 concludes the paper highlighting also some possible future work.

## 2 System model and problem statement

We assume that the system is composed by a number of interacting clients that can act as publishers (data producers) or subscribers (data consumers). Clients can exchange data using a topic-based publish/subscribe interface, thus we also assume that they share a common knowledge on a set of available topics. Each piece of data produced by a publisher is published on one of the available topics and takes the form of an event. Each subscriber issues a subscription containing the set of topics it is interested in. An event  $e$  published on a topic  $T$  matches a subscription  $S$  if and only if  $T \in S$ ; if this happens, the corresponding subscriber must be notified about  $e$ . Clients do not interact directly; their interactions are mediated by an *Event Notification Service* (ENS) that exposes the fundamental interface of a publish/subscribe system, i.e., the *publish*, *subscribe/unsubscribe* and *notify* primitives. We assume that the ENS is implemented as distributed middleware.

In addition, in order to simplify the description of our solution, we will initially assume that our system works on top of a reliable communication substrate, that all communication links deliver messages in FIFO order, and that all processes are correct. Some of these assumptions will be removed or relaxed in Section 5.

The ordering property we want to enforce is defined as follows:

**Property 1** TOTAL NOTIFICATION ORDER (TNO). *Let  $e_i$  and  $e_j$  be two distinct events notified to a subscriber  $s$ . If  $e_i$  is notified to  $s$  before  $e_j$ , no subscriber will be notified about  $e_i$  after being notified about  $e_j$ .*

Note that this definition matches the definition of *Weak Total Order* given in [1] in the context of total order specifications [9]. Differently from those specifications, we do not consider any form of deterministic agreement (uniform or not uniform) because here we are only interested in designing an ordering layer to be transparently plugged on top of a generic ENS which can provide different reliability and agreement properties.

Guaranteeing TNO in a distributed setting is a non trivial task. Consider, for example, the toy system depicted in Figure 1: the six black dots represent processes constituting the ENS, the white dots on the left ( $p_1$  and  $p_2$ ) are two publishers and those on the right ( $s_1$  and  $s_2$ ) are two subscribers.

A simple solution for ordering events published in a specific topic is based on the usage of topic managers: a single node in the ENS is elected as a “sequencer” for all the events published in that topic. In our example

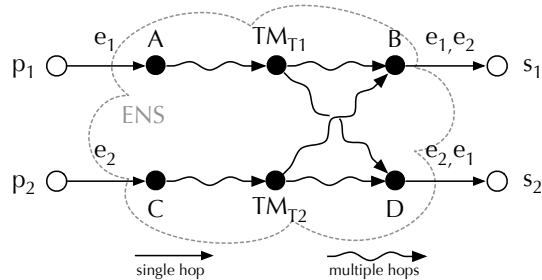


Figure 1: An example showing how notifications can be performed out of order in a distributed event notification service.

$TM_{T_1}$  acts as the topic manager node for topic  $T_1$ , receiving all the events published in  $T_1$  (i.e., event  $e_1$  published by  $p_1$ ), adding a sequence number to them, and then routing the events toward the intended destinations (i.e.,  $s_1$  and  $s_2$  notified by nodes  $B$  and  $D$ ).

However, this simple approach is not useful when the subscriptions intersect in multiple topics. For example, assume that both  $s_1$  and  $s_2$  are subscribed to  $T_1$  and  $T_2$ . In this case the sequence numbers attached by  $TM_{T_1}$  and  $TM_{T_2}$  would be completely uncorrelated and thus useless to check for a correct notification order on the subscribers' side. The obvious solution, i.e. having a single topic manager for all the topics, has important scalability and reliability drawbacks and cannot thus be considered as a realistic alternative.

### 3 The event ordering algorithm

This Section first describes how the proposed solution can fit within an existing architecture based on publish/subscribe interactions, and then, details the algorithm that implements the solution.

#### 3.1 Architectural aspects

Our solution assumes that all participants to the system (publishers and subscribers) are equipped with an *Ordering module* that implements the algorithm described in the next Section (see Figure 2). This module mediates the interactions between application level software components, that act as information producers (publisher applications) or consumers (subscriber applications), and a standard ENS.

The only assumption we do on the ENS is that it implements a standard topic-based publish/subscribe interface (here represented by the *ENSpublish*, *ENSsubscribe/ENSunsubscribe* and *ENSnotify* operations). The same

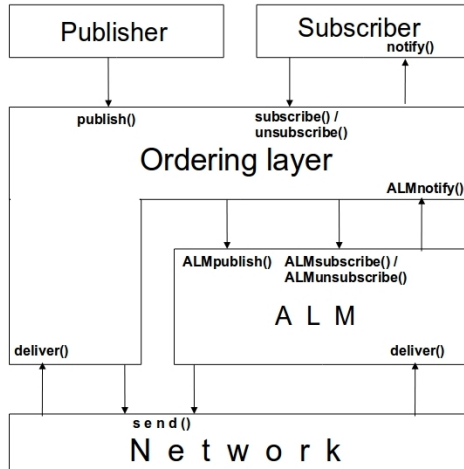


Figure 2: Architectural view that shows how the ordering module acts as a mediating software layer between the applications and an existing event notification service.

interface is offered by the ordering module to the application level, therefore neither the applications, nor the ENS must be changed in order to work with our module. The ordering module also needs to access a point-to-point communication primitive that can be offered by the operating system or by other solutions like an overlay network. We also assume that the set of available topics is fixed and a precedence relationship  $\rightarrow$  holds among topic identifiers inducing a total order on them<sup>3</sup>. Moreover, we assume that there is a method to univocally map a topic  $T$  to a single participating node in the system that will act as the *topic manager* for that topic ( $TM_T$ ). This latter assumption can be satisfied in several different ways, e.g. through a static mapping provided as a configuration parameter or using a distributed hash table as in rendez-vous based publish/subscribe systems [7]. In the following, whenever there is no ambiguity, we will use the terms *publisher* and *subscriber* to refer the parts of our ordering module located respectively at the publisher and at the subscriber site.

### 3.2 Algorithm description

The basic idea behind the algorithm is to assign a logical timestamp to each event. By looking at a timestamp, a subscriber must be able to decide if the event can be notified or it needs a tag, witnessing that it was received out of order. The notified application will then take a decision on how out-of-order events must be treated (i.e., discarded). The algorithm to be executed

<sup>3</sup>This assumption will be relaxed in Section 5 where we will show how this order can be changed at run-time in order to improve the performance.

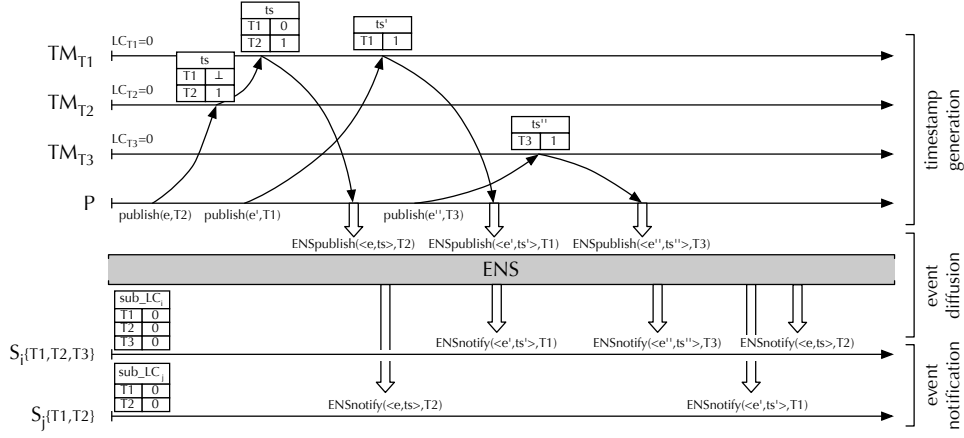


Figure 3: Example of a system run with two subscribers,  $S_i$  and  $S_j$ , and a publisher  $P$ .

when an event is published is split in three phases: (i) timestamp generation, where a timestamp is generated for the event, (ii) event diffusion, where the ENS delivers the event and its timestamp to all the intended subscribers, and (iii) event notification, where subscribers, by looking at the timestamp content, decide if the event must be tagged as out-of-order before notifying it (Figure 3). The algorithm uses only local information maintained by each process: a topic manager  $TM_T$  stores all subscriptions containing topic  $T$  and a sequence number that counts the number of events published on  $T$ , while each subscriber stores its subscription  $S$  and a set containing the sequence number of the last event notified on  $T$ , for each topic  $T \in S$  (i.e. it maintains a local subscription clock).

The first phase is started by a publisher requiring the creation of a timestamp for a new event  $e$  published on a topic  $T$ . The timestamp creation for  $e$  is carried out by the subset of  $TM$ s associated to topics belonging to the *sequencing group* of  $T$  (namely  $\mathcal{SG}_T$ ).  $\mathcal{SG}_T$  is a set of topics including all  $T'$  such that (i)  $T' \rightarrow T$  and (ii) there are at least two subscriptions including both  $T$  and  $T'$ . The timestamp generation procedure is started by  $TM_T$  when it receives the request of a timestamp generation from the publisher.  $TM_T$  creates the structure of the timestamp adding one entry for each topic  $T'$  such that  $T' \in \mathcal{SG}_T$ , stores  $T$ 's current sequence number and forwards the timestamp to the  $TM$  associated to the first topic in  $\mathcal{SG}_T$  that precedes  $T$  according to the precedence relation  $\rightarrow$ . Note that, given a specific order  $T_1 \rightarrow T_2 \rightarrow \dots \rightarrow T_n$  among topics, the timestamp generation flow proceeds in the opposite direction (i.e. given a topic  $T_i$ ,  $TM_{T_i}$  will fill in the timestamp and forward it to some  $TM_{T_j}$  such that  $T_j \rightarrow T_i$ ). The receiving  $TM$  adds the sequence number for the topic it manages to the timestamp and sends it to the  $TM$  associated to the next topic in  $\mathcal{SG}_T$ .

When the last  $TM$  completes the timestamp, it is returned to the publisher that will publish the event on the ENS together with the timestamp, starting the event diffusion phase. This collaboration among several  $TMs$  in the creation of a timestamp is fundamental to totally order events published on their corresponding topics, and thus it avoids possible TNO violations like the one shown at the end of Section 2. Referring to the example in Figure 3, where the topic order is  $T_1 \rightarrow T_2 \rightarrow T_3$ , the publisher  $P$  publishes an event  $e$  on topic  $T_2$  and asks  $TM_{T_2}$  to create the timestamp. In this case  $\mathcal{SG}_{T_2} = \{T_2, T_1\}$ . Therefore  $TM_{T_2}$  creates the structure of the timestamp with entries for topics  $T_2$  and  $T_1$ , puts its sequence number in the timestamp and forwards it to  $TM_{T_1}$  that, in turn, will complete the timestamp and return it to  $P$ . Finally  $P$  publishes both  $e$  and its timestamp on the ENS.

In the event notification phase, once an event  $e$  and its timestamp are notified by the ENS, the subscriber checks if the timestamp attached to the event is coherent with the event order maintained through the local subscription clock. If so, the event is notified to the application, otherwise it is tagged to let the application be aware that it is notified out-of-order (cfr. paragraph NOTIFY() Operation in the following). Once an application is notified about an unordered event, it will decide, according to its specific requirements, if the order inversion can be tolerated or if the event must be discarded.

Note that, given a topic  $T$ , its sequencing group is defined according to all subscriptions containing  $T$ . As a consequence, every new subscription (or unsubscription) to topic  $T$  induces a modification of the subset of topic managers involved in the timestamp generation phase. Thus, when a topic  $T$  is added to a subscription  $S$ , each topic manager  $TM_{T'}$  associated to a topic  $T'$  in  $S$  must be advertised about the change of the subscription to avoid possible TNO violation. To this aim, the subscriber creates an empty subscription timestamp, containing one entry for each topic in the subscription. Similarly to event timestamp, each entry of the subscription timestamp is filled in by the corresponding topic manager. In addition, receiving the request, each topic manager  $TM_T$  updates the list of subscriptions it knows. When the sequence number of the last topic manager belonging to the subscription is added, the latter sends the complete timestamp to the subscriber, that, in turn, resets its local subscription clock. The same approach is used to unsubscribe a topic. When a subscriber is no longer interested in events of a topic  $T$ , it advertises the change on the subscription to topic managers managing topics in its subscription. Receiving the unsubscription, a topic manager just updates the set of subscriptions it knows.

In the following, before describing the algorithm operations in detail, we first provide the definition of a *timestamp associated to an event  $e$*  and then we specify an *order relation among two timestamps*.

**Definition 1** Let  $e$  be an event published on a topic  $T$ ,  $T_1 \rightarrow T_2 \rightarrow \dots \rightarrow T_n$  be the topic ordering and let  $\mathcal{SG}_T$  be the sequencing group of  $T$ . A timestamp  $ts_e$  for  $e$  is a set of pairs  $\langle T_i, sn_i \rangle$  ordered according to the precedence relation  $\rightarrow$ , where  $T_i \in \mathcal{SG}_T$  is a topic identifier and  $sn_i$  is  $e$ 's sequence number for topic  $T_i$ .

**Definition 2** Let  $ts_e$  and  $ts_{e'}$  be two timestamps associated with two different events  $e$  and  $e'$ . We say that  $ts_e$  and  $ts_{e'}$  are comparable if there exists at least one topic identifier included both in  $ts_e$  and  $ts_{e'}$  (i.e.  $\exists t_{id}, i, j \mid (\langle t_{id}, i \rangle \in ts_e) \wedge (\langle t_{id}, j \rangle \in ts_{e'})$ ).

From the two definitions above, it is easy to see that given two events  $e$  and  $e'$  published respectively on topic  $T$  and  $T'$ , the corresponding timestamps  $ts_e$  and  $ts_{e'}$  are comparable if and only if  $\mathcal{SG}_T \cap \mathcal{SG}_{T'} \neq \emptyset$ .

**Definition 3** Let  $ts_e$  and  $ts_{e'}$  be two timestamps associated with two different events  $e$  and  $e'$ . We say that  $ts_e$  is smaller than  $ts_{e'}$  (i.e.  $ts_e < ts_{e'}$ ) if

1.  $ts_e$  and  $ts_{e'}$  are comparable and
2.  $\forall \langle t_{id}, sn \rangle \in ts_e \mid \exists \langle t_{id}, sn' \rangle \in ts_{e'}, sn \leq sn'$  and
3.  $\exists \langle t_{id}, sn \rangle \in ts_e \mid \exists \langle t_{id}, sn' \rangle \in ts_{e'}, sn < sn'$

As an example, in Figure 3 we show the timestamps for three published events  $e$ ,  $e'$  and  $e''$ . Considering the timestamp  $ts$  associated to  $e$  and the timestamp  $ts'$  associated to  $e'$  we have that they are comparable (there is at least one topic, i.e.,  $T_1$ , belonging to both  $ts$  and  $ts'$ ); on the contrary,  $ts$  and  $ts''$  (or even  $ts'$  and  $ts''$ ) are not comparable. Moreover, considering  $ts$  and  $ts'$ , we have that  $ts < ts'$ .

**Local data structures to each publisher  $p_i$ :** each publisher maintains locally the following data structures:

- $id_e$ : is a unique identifier associated to each event produced by  $p_i$ .
- $outgoingEvents_i$ : a set variable, initially empty, storing the events indexed by event id that are published by the upper application layer, and that are waiting for being published on the ENS.

**Local data structures to each subscriber  $s_i$ :** each subscriber maintains locally the following data structures:

- $subs_i$ : a set variable storing topics subscribed by  $p_i$ ;

- $sub\_LC_i$ : a set of pairs  $\langle T_i, sn_i \rangle$ , where  $T_i$  is a topic identifier and  $sn_i$  is an integer value;  $sub\_LC_i$  contains a pair for each topic  $T_i \in subs_i$ . Initially, for each topic  $T_i \in subs_i$  the corresponding sequence number is  $\perp$ .

**Local data structures to each topic manager  $TM_{T_i}$ :** to simplify the notation, we assume that each topic manager  $TM_i$  is responsible for one topic  $T_i$ <sup>4</sup>. Each topic manager maintains locally the following data structures:

- $LC_{T_i}$ : is an integer value representing the sequence number associated to topic  $T_i$ , initially 0.
- $externalSubs_{T_i}$ : a set of pairs  $\langle id, sub \rangle$  where  $sub$  is a subscription (i.e. a set of topics  $\{T_j, T_k \dots T_h\}$ ) and  $id$  is the subscriber identifier. Such a set contains all the subscriptions that include  $T_i$ .

As an example, let us consider the system depicted in Figure 3. Let  $S_i = \{T1, T2, T3\}$  and  $S_j = \{T1, T2\}$  be respectively the two subscriptions of  $s_i$  and  $s_j$ . The three variables  $externalSubs$  maintained by each topic manager are respectively:  $externalSubs_{T1} = \{\langle i, S_i \rangle, \langle j, S_j \rangle\}$ ,  $externalSubs_{T2} = \{\langle i, S_i \rangle, \langle j, S_j \rangle\}$  and  $externalSubs_{T3} = \{\langle i, S_i \rangle\}$ .

**PUBLISH() Operation.** The algorithm for a PUBLISH() operation is shown in Figure 5. To simplify the pseudo-code of the algorithm, we defined the following basic functions:

- $generateUniqueEventID(e)$ : generates a locally unique identifier for a specific event  $e$ .
- $next(ts, T)$ : given a timestamp  $ts$  and a topic identifier  $T$ , the function returns the identifier of the topic  $T'$  preceding  $T$  in the timestamp  $ts$ , according to the precedence relation  $\rightarrow$ ; if a *null* value is passed as topic identifier, the function returns the last topic identifier contained in the timestamp.
- $getTopicRespAddress(T)$ : returns the network address of the topic manager  $TM_T$  responsible for topic  $T$ .
- $update(ts, T, LC_T)$ : updates the event timestamp  $ts$  changing the pair  $\langle T, - \rangle$  with the pair  $\langle T, LC_T \rangle$ .

---

<sup>4</sup>This assumption does not limit the applicability of our solution. If the topic manager is responsible for more than one topic, its local variables must be duplicated, one for each topic.

In addition, we have defined a more complex function, namely `createPubTimestamp` ( $T, externalSubs_T$ ), that generates an empty timestamp for a generic event published on topic  $T$  by considering the set of subscriptions containing  $T$  (i.e. subscriptions stored in  $externalSubs_T$ ). The pseudo-code of the function is shown in Figure 4.

```

function createPubTimestamp( $T, externalSubs$ ):
(01) for each  $\langle -, s \rangle \in externalSubs$  do  $SG_T \leftarrow SG_T \cup s$ ; endfor
(02) sort ( $SG_T$ );
(03) for each  $T_j \in SG_T : T_i \rightarrow T_j$  do  $SG_T \leftarrow SG_T / \{T_j\}$ ; endfor
(04) for each  $T_j \neq T_i \in SG_T$  do
(05)   let  $S = \{s \in externalSubs | T_j \in s\}$ ;
(06)   if ( $|S| \leq 1$ ) then  $SG_T \leftarrow SG_T / \{T_j\}$  endif
(07) endfor
(08) for each  $T_j \in SG_T$  do  $ts \leftarrow ts \cup \{\langle T_j, \perp \rangle\}$ ; endfor
(09) return  $ts$ .

```

Figure 4: The `createPubTimestamp()` function (for a topic manager  $TM_{T_i}$ ).

We want to remark that an event timestamp has an entry only for those topics that precede  $T$  in the topic order (line 03) and that appear in more than one subscription together with  $T$  (lines 04-07).

Considering the execution depicted in Figure 3 and the topic order  $T_1 \rightarrow T_2 \rightarrow T_3$ , we show how the timestamp of the event  $e$  published on  $T_2$  is created. The procedure can be summarized in the following steps: (i)  $SG_{T_2} = \{T_1, T_2, T_3\}$  initially represents the sorted union of all the subscriptions containing  $T_2$  (lines 01-02), (ii)  $SG_{T_2} = \{T_1, T_2\}$  is the result after filtering out topics following  $T_2$  according to the precedence relation  $\rightarrow$  (line 03), and (iii)  $ts_e = \{\langle T_1, \perp \rangle, \langle T_2, \perp \rangle\}$  is the empty timestamp built from  $SG_{T_2}$ .

When an event  $e$  is published on a topic  $T$ , the publisher  $p_i$  executes the algorithm shown in Figure 5(a). In particular, it associates to  $e$  a unique identifier generated locally (line 01), it puts the event, together with the topic and the corresponding identifier in a buffer (line 02) and sends a `CREATE_PUB_TS` ( $id_e, i, T$ ) message to the topic manager  $TM_T$ , associated to  $T$  (lines 03-04).

Receiving the `CREATE_PUB_TS` ( $id_e, i, T$ ) message,  $TM_T$  executes the algorithm shown in Figure 5(b). In particular, it first creates an empty timestamp  $ts_e$  by executing the `createPubTimestamp` function, it increments its local sequence number (line 02), updates the corresponding entry in  $ts_e$  (line 03) and sends a `FILL_IN_PUB_TS` message containing the timestamp, to the preceding topic manager until  $ts_e$  has been completed and it is finally returned to the publisher. Note that, when a topic manager receives a `FILL_IN_PUB_TS` message, it just attaches its local sequence number (line 12). Figure 3 shows an example of the complete publish procedures for three different events with the corresponding timestamps.

```

operation PUBLISH( $e, T$ ):
(01)  $id_e \leftarrow \text{generateUniqueEventID}(e)$ ;
(02)  $outgoingEvents \leftarrow (e, id_e, T)$ ;
(03)  $dest \leftarrow \text{getTopicRespAddress}(T)$ ;
(04) send CREATE_PUB_TS( $id_e, i, T$ ) to  $dest$ ;
-----
(05) when EVENT_TS( $ts, e_{id}$ ) is delivered:
(06)   let  $\langle e, id_e, T \rangle \in outgoingEvents$ 
(07)   such that ( $e_{id} = id_e$ );
(08)   ENSpublish( $\langle e, ts \rangle, T$ );
(09)    $outgoingEvents \leftarrow outgoingEvents / \{\langle e, id_e, T \rangle\}$ .

```

(a) Publisher Protocol (for a publisher process  $p_i$ )

```

(01) when CREATE_PUB_TS( $e_{id}, j, T$ ) is delivered:
(02)    $ev\_ts \leftarrow \text{createPubTimestamp}(T, externalSubs_{T_i})$ ;
(03)    $LC_{T_i} \leftarrow LC_{T_i} + 1$ ;
(04)    $ev\_ts \leftarrow \text{update}(ev\_ts, \langle T_i, LC_{T_i} \rangle)$ ;
(05)   if ( $|ev\_ts| = 1$ )
(06)     then send EVENT_TS( $ev\_ts, e_{id}$ ) to  $p_j$ ;
(07)     else  $t' \leftarrow \text{next}(ev\_ts, T_i)$ ;
(08)        $dest \leftarrow \text{getTopicRespAddress}(t')$ ;
(09)       send FILL_IN_PUB_TS( $ev\_ts, e_{id}, j$ ) to  $dest$ ;
(10)   endif
-----
(11) when FILL_IN_PUB_TS( $ts, e_{id}, j$ ) is delivered:
(12)    $ts \leftarrow \text{update}(ts, \langle T_i, LC_{T_i} \rangle)$ ;
(13)    $t' \leftarrow \text{next}(ts, T_i)$ ;
(14)   if ( $t' = null$ )
(15)     then send EVENT_TS( $ts, e_{id}$ ) to  $p_j$ ;
(16)     else  $dest \leftarrow \text{getTopicRespAddress}(t')$ ;
(17)       send FILL_IN_PUB_TS( $ts, e_{id}, j$ ) to  $dest$ ;
(18)   endif.

```

(b) TM Protocol (for a topic manager  $TM_{T_i}$ )

Figure 5: The publish() protocol.

**NOTIFY() Operation.** When an event  $e$  is notified by the ENS, a subscriber  $s_i$  executes the algorithm shown in Figure 6. The algorithm uses a function  $\text{tag}(e)$  that creates a new event  $e'$ , containing  $e$  and the indication that it has been delivered out-of-order. The event  $e$  is not immediately notified to the application layer. A subscriber  $s_i$  first checks if the event has been published on a topic actually subscribed by  $s_i$  and then checks if it has been notified by the ENS in the right order (line 01). If such condition is not satisfied, a new event  $e'$  is created by tagging  $e$ ; then, the event  $e'$  is notified to the application (lines 09-11).

On the contrary, if the event can be notified,  $s_i$  triggers the notification to the application (line 02) and then updates its local subscription clock with the sequence numbers contained in the event timestamp (lines 03-08).

**SUBSCRIBE() and UNSUBSCRIBE() Operations.** The algorithm for a SUBSCRIBE() operation is shown in Figure 7. To simplify the pseudo-code of the algorithm,

```

upon ENSnotify( $\langle e, ts \rangle, T$ ):
(01) if ( $(T \in subs_i) \wedge (sub\_LC_i < ts)$ )
(02)   then trigger notify ( $e, T$ );           % ordered notification
(03)     for each  $\langle T_j, v \rangle \in sub\_LC_i$ 
(04)       if ( $(\exists \langle T_j, v' \rangle \in ts) \wedge (v' > v)$ )
(05)         then  $sub\_LC_i \leftarrow sub\_LC_i / \{ \langle T_j, v \rangle \}$ 
(06)            $sub\_LC_i \leftarrow sub\_LC_i \cup \{ \langle T_j, v' \rangle \}$ 
(07)       endif
(08)     endfor
(09)   else  $e' \leftarrow \text{tag}(e)$ ;
(10)     trigger notify ( $e', T$ );           % out-of-order notification
(11) endif

```

Figure 6: The notify() protocol (for subscriber  $s_i$ ).

```

operation SUBSCRIBE( $T$ ):
(01)  $ts \leftarrow \text{createSubTimestamp}(subs_i \cup T)$ ;
(02)  $t' \leftarrow \text{next}(ts, null)$ ;
(03)  $dest \leftarrow \text{getTopicRespAddress}(t')$ ;
(04) send FILL_IN_SUB_TS ( $ts, (subs_i \cup \{T\}), id$ ) to  $dest$ ;


---


(05) when COMPLETED_SUB_VC ( $ts, s$ ) is delivered:
(06)    $sub\_LC_i \leftarrow ts$ ;
(07)    $subs_i \leftarrow subs_i \cup \{T\}$ ;
(08)   ENSsubscribe( $T$ );

```

(a) Subscriber Protocol

```

(01) when FILL_IN_SUB_TS ( $ts, sub, j$ ) is delivered:
(02)    $externalSubs_i \leftarrow \text{update}(externalSubs_i, \langle j, sub \rangle)$ ;
(03)    $LC_{T_i} \leftarrow LC_{T_i} + 1$ 
(04)    $ts \leftarrow \text{update}(ts, \langle T_i, LC_{T_i} \rangle)$ ;
(05)    $t' \leftarrow \text{next}(ts, T_i)$ ;
(06)   if ( $t' = null$ ) then send COMPLETED_SUB_VC ( $ts, s$ ) to  $j$ ;
(07)     else  $dest \leftarrow \text{getTopicRespAddress}(t')$ ;
(08)       send FILL_IN_SUB_TS ( $ts, s, j$ ) to  $dest$ ;
(09)   endif

```

(b) TM Protocol

Figure 7: The subscribe() protocol.

in addition to the functions used in the PUBLISH() algorithm, we defined the createSubTimestamp( $sub$ ) function, that creates an empty subscription timestamp, i.e., a set of pairs  $\langle T, sn \rangle$  where  $T$  is a topic identifier and  $sn$  is the sequence number for  $T$ , initially set to  $\perp$ . The subscription timestamp contains a pair for each topic  $T$  of a subscription  $S$ .

When a subscriber  $s_i$  wants to subscribe a new topic  $T$ , it executes the algorithm shown in Figure 7(a). In particular, it creates an empty subscription timestamp through the createSubTimestamp function (including also topic  $T$ ), and then it sends a FILL\_IN\_SUB\_TS ( $ts, (subs_i \cup \{T\}), id$ ) message to fill the timestamp and to forward the new subscription to the topic manager  $TM_{T_k}$  responsible for the last topic in the subscription, according to

the precedence relation  $\rightarrow$  (lines 02-04).

```

operation UNSUBSCRIBE( $T, subID$ ):
(01)  $subs_i \leftarrow subs_i / \{T\}$ ;
(02) for each  $T_j \in subs_i$  do
(03)    $dest \leftarrow \text{getTopicRespAddress}(T_j)$ ;
(04)   send UPDATE_SUB ( $subID, subs_i$ ) to  $dest$ ;
(05) endfor
(06)  $dest \leftarrow \text{getTopicRespAddress}(T)$ ;
(07) send UPDATE_SUB ( $subID, \emptyset$ ) to  $dest$ ;
(08) ENSunsubscribe( $T$ );

```

(a) Subscriber Protocol

```

when UPDATE_SUB ( $id, s$ ) is delivered:
(01) if ( $s \neq \emptyset$ )
(02)   then  $externalSubs_i \leftarrow \text{update}(externalSubs_i, \langle id, s \rangle)$ ;
(03)   else  $externalSubs_i \leftarrow externalSubs_i / \{\langle id, - \rangle\}$ ;
(04) endif.

```

(b) TM Protocol

Figure 8: The unsubscribe() protocol.

Upon the delivery of a FILL\_IN\_SUB\_TS message, each topic manager  $TM_{T'}$  executes the algorithm shown in Figure 7(b). In particular,  $TM_{T'}$  updates its  $externalSubs_k$  variable with the new subscription (line 02), increments its local sequence number (line 03), updates its entry in the subscription timestamp (line 04) and finally forwards the FILL\_IN\_SUB\_TS message to the preceding topic manager until it is completed and returned to the client. When the subscriber receives the completed subscription timestamp, it updates its local subscription clock (line 06) and then makes the subscription effective by calling the ENSsubscribe() method (line 08).

The algorithm for the UNSUBSCRIBE() operation is shown in Figure 8. A subscriber that wants to unsubscribe from a topic  $T$ , removes it from the set of subscribed topics (line 01) and, then, informs all topic managers of these topics with the updated subscription through an UPDATE\_SUB message (lines 02-05), including the topic manager of  $T$  that will receive an empty subscription (lines 06-07). When receiving an UPDATE\_SUB message (Figure 8(b)), topic managers update the  $externalSubs$  set accordingly with the received subscription.

### 3.3 Correctness proof

In this Section, we will show that the TNO property holds for any pair of non-tagged events.

**Definition 4** Given a generic subscriber  $s_i$ , let us denote  $\tau_n(i, e)$  as the time instant at which the ENS notifies an event  $e$  to  $s_i$ .

**Lemma 1** *Let  $e_1$  and  $e_2$  be two events published respectively on a topic  $T$ . If a subscriber  $s_i$  notifies  $e_1$  before  $e_2$  then any other subscriber that notifies both  $e_1$  and  $e_2$  will notify  $e_1$  before  $e_2$ .*

**Proof** Let us suppose by contradiction that there exist two subscribers, namely  $s_i$  and  $s_j$ , that notify both  $e_1$  and  $e_2$  published on topic  $T$ , but  $s_i$  notifies  $e_1$  and then  $e_2$  (i.e.,  $\tau_n(i, e_1) < \tau_n(i, e_2)$ ), while  $s_j$  notifies  $e_2$  and then  $e_1$  (i.e.,  $\tau_n(j, e_2) < \tau_n(j, e_1)$ ).

Given a generic subscriber  $s_k$  notifying both  $e_1$  and  $e_2$ , it follows that at time  $\tau_n(k, e_1)$ ,  $T_1 \in S_k$  and at time  $\tau_n(k, e_2)$ ,  $T_2 \in S_k$ .

Moreover,  $sub\_LC_k(\tau_n(k, e_1)) \leq ts_{e_1}$  and  $sub\_LC_k(\tau_n(k, e_2)) \leq ts_{e_2}$ .

Given the two timestamps  $ts_{e_1}$  and  $ts_{e_2}$  associated respectively to  $e_1$  and  $e_2$ , let us first consider how they have been created and then let us show that it is not possible to have inversions in the notification order.

Let  $p_k$  and  $p_h$  be respectively the publishers of events  $e_1$  and  $e_2$ . When a publisher publishes an event, it executes line 04 of Figure 5(a) and it sends a CREATE\_PUB\_TS message.

Without loss of generality, let us assume that  $TM_T$  delivers first the CREATE\_PUB\_TS message sent by  $p_k$  and then the CREATE\_PUB\_TS message sent by  $p_h$ .

Let  $v$  be the value of the local clock maintained by  $TM_T$  when it delivers the CREATE\_PUB\_TS message sent by  $p_k$ . When  $TM_T$  delivers such a message, it creates an empty event timestamp  $ts_{e_1}$  through the execution of the createPubTimestamp function (cfr. Figure 4), it increments its local clock (i.e.  $LC_T = v + 1$ ) and it includes the pair  $\langle T, v + 1 \rangle$  in  $ts_{e_1}$  (lines 03 - 04, Figure 5(b)). Then two cases can happen:

1.  $ts_{e_1}$  **contains only the entry for  $T$  (line 05)**:  $ts_e = \{\langle T, v + 1 \rangle\}$  and  $TM_T$  returns the completed timestamp to the publisher for the publication on the ENS.
2.  $ts_{e_1}$  **contains more than one entry (lines 06 - 09)**: in this case, there exists a topic  $T'$  following  $T$  in the topic order and  $TM_T$  sends a FILL\_IN\_PUB\_TS message to  $TM_{T'}$ . Receiving such a message,  $TM_{T'}$  just updates the pair  $\langle T', \perp \rangle$  contained in  $ts_e$  with its current sequence number for  $T'$  and it checks if there exists a topic  $T''$  following  $T'$  in the timestamp. If so, it forwards the FILL\_IN\_PUB\_TS message to  $TM_{T''}$ , otherwise, it returns  $ts_e$  to the publisher (lines 11 - 16).

When  $TM_T$  delivers the CREATE\_PUB\_TS message sent by  $p_h$ , it repeats the previous actions: it creates a template  $ts_{e_2}$  for the timestamp, increments its local sequence number (i.e.  $LC_T = v + 2$ ), includes the pair  $\langle T, v + 2 \rangle$  in  $ts_{e_2}$  and sends the timestamp to the publisher or to the following topic manager.

Considering that the subscriptions are stable, the timestamp will always include the same entries, i.e.,  $ts_{e_1}$  and  $ts_{e_2}$  contain a set of pairs differing only for the sequence numbers associated to each topic. In particular, considering that (i) a topic manager can only increment its local sequence number when a publish event occurs, and (ii) topic managers are connected through FIFO channels, it follows that for each topic  $T_i$  the sequence number  $v'$  associated to  $T_i$  in  $ts_{e_2}$  cannot be smaller than the one associated to  $T_i$  in  $ts_{e_1}$ . Therefore,  $ts_{e_1} < ts_{e_2}$ .

Considering that (i) as soon as an event  $e$  is notified to the application layer, the local subscription clock of the subscriber is updated according to the event timestamp (lines 03 - 06, Figure 6), and that (ii)  $ts_{e_1} < ts_{e_2}$ , we have that  $s_j$  evaluating the notification condition at line 01 will discard event  $e_1$  after the notification of  $e_2$ . This clearly leads to a contradiction.

□*Lemma 1*

**Lemma 2** *Let  $e_1$  and  $e_2$  be two events published respectively on a topic  $T_1$  and on a topic  $T_2$ , with  $T_1 \neq T_2$ . If a subscriber  $s_i$  notifies  $e_1$  before  $e_2$  then any other subscriber that notifies both  $e_1$  and  $e_2$  will notify  $e_1$  before  $e_2$ .*

**Proof** For ease of presentation and without loss of generality, let us assume that  $T_1$  and  $T_2$  are the two only topics subscribed by both  $s_i$  and  $s_j$ <sup>5</sup> (i.e.  $\{T_1, T_2\} \subseteq S_i, S_j$ ).

Let us suppose by contradiction that there exist two subscribers, namely  $s_i$  and  $s_j$ , that notify both  $e_1$  (published on topic  $T_1$ ) and  $e_2$  (published in topic  $T_2$ ) but  $s_i$  notifies  $e_1$  and then  $e_2$  (i.e.  $\tau_n(i, e_1) < \tau_n(i, e_2)$ ) while  $s_j$  notifies  $e_2$  and then  $e_1$  (i.e.  $\tau_n(j, e_2) < \tau_n(j, e_1)$ ).

Given a generic subscriber  $s_k$ , if it notifies both  $e_1$  and  $e_2$ , it follows that, at time  $\tau_n(k, e_1)$ ,  $T_1 \in S_k$  and at time  $\tau_n(k, e_2)$ ,  $T_2 \in S_k$ . Moreover,  $sub\_LC_k(\tau_n(k, e_1)) \leq ts_{e_1}$  and  $sub\_LC_k(\tau_n(k, e_2)) \leq ts_{e_2}$ .

Given the timestamps  $ts_{e_1}$  and  $ts_{e_2}$  associated respectively to  $e_1$  and  $e_2$ , let us first consider how they have been created and then let us show that it is not possible to have inversions in the notification order.

Without loss of generality, let us assume that  $T_1$  has higher priority than  $T_2$  in the topic order (i.e., in any timestamp containing an entry for both  $T_1$  and  $T_2$ ,  $T_1$  follows  $T_2$  in the sequence). Considering the `createPubTimestamp` function shown in Figure 4, each event published on  $T_1$  will have attached a timestamp containing a pair  $\langle T_1, v_1 \rangle$  and each event published on  $T_2$  will have attached a timestamp containing the following pairs  $\langle T_1, v_1 \rangle, \langle T_2, v_2 \rangle$ .

When  $e_2$  is published by the application layer, the publisher sends a `CREATE_PUB_TS` request for the event timestamp to  $TM_{T_2}$  (line 04, Figure

---

<sup>5</sup>The proof can be easily extended to multiple intersections, by iterating the reasoning for any pair of topics that appears in more than one subscription.

5(a)). Receiving such a request,  $TM_{T_2}$  executes line 01 of Figure 5(b) (i.e. `createPubTimestamp` function) and creates an empty event timestamp containing an entry both for  $T_1$  and  $T_2$  (i.e.,  $ts_{e_2} \supseteq \langle T_1, \perp \rangle, \langle T_2, \perp \rangle$ ), it increments its local clock, let's say to a value  $v$  (line 02), it updates its component of the timestamp with its local clock (i.e.  $ts_{e_2} \supseteq \langle T_1, \perp \rangle, \langle T_2, v \rangle$ ) and then it sends a `FILL_IN_PUB_TS` message containing  $ts_{e_2}$  to the following topic manager selected in the event timestamp according to the precedence relation  $\rightarrow$  (i.e., to  $TM_{T_1}$ ).

The same procedure is executed when  $e_1$  is published.

Note that, in the worse case scenario, due to concurrency in the timestamp creation procedure,  $TM_{T_1}$  can either deliver first the `FILL_IN_PUB_TS` message sent by  $TM_{T_2}$  and then the `CREATE_PUB_TS` sent from the publisher of  $e_2$  or viceversa, it can first manage the `CREATE_PUB_TS` message and then the `FILL_IN_PUB_TS` message.

1.  **$TM_{T_1}$  delivers the `CREATE_PUB_TS` message for event  $e_1$  and then the `FILL_IN_PUB_TS` message for  $ts_{e_2}$ .** Delivering the `CREATE_PUB_TS` for event  $e_1$ ,  $TM_{T_1}$  creates an empty event timestamp for  $e_1$  (i.e.,  $ts_{e_1} \supseteq \langle T_1, \perp \rangle$ ), it updates its local clock to  $v_1 + 1$ , it updates its component of the timestamp with its local clock (i.e.,  $ts_{e_1} \supseteq \langle T_1, v_1 + 1 \rangle$ ) and then it sends a `FILL_IN_PUB_TS` request containing  $ts_{e_1}$  to the following topic manager in the topic order (if any) or directly to the publisher.

Delivering the `FILL_IN_PUB_TS` message for  $ts_{e_2}$ ,  $TM_{T_1}$  executes line 11 of Figure 5(b), and updates its component of the timestamp with its local clock (i.e.,  $ts_{e_2} \supseteq \langle T_1, v_1 + 1 \rangle, \langle T_2, v \rangle$ ). Then, it sends a `FILL_IN_PUB_TS` request containing  $ts_{e_2}$  to the following topic manager in the topic order (if any) or directly to the publisher.

2.  **$TM_{T_1}$  delivers the `FILL_IN_PUB_TS` message for  $ts_{e_2}$  and then the `CREATE_PUB_TS` message for event  $e_1$ .** Delivering the the `FILL_IN_PUB_TS` message for  $ts_{e_2}$ ,  $TM_{T_1}$  executes line 11 of Figure 5(b), and updates its component of the timestamp with its local clock (i.e.,  $ts_{e_2} \supseteq \langle T_1, v_1 \rangle, \langle T_2, v \rangle$ ). Then, it sends a `FILL_IN_PUB_TS` request containing  $ts_{e_2}$  to the following topic manager in the topic order. On the contrary, delivering the `CREATE_PUB_TS` for event  $e_1$ ,  $TM_{T_1}$  creates the template for the event timestamp (i.e.,  $ts_{e_1} \supseteq \langle T_1, \perp \rangle$ ), updates its local clock to  $v_1 + 1$ , updates its component of the timestamp with its local clock (i.e.,  $ts_{e_1} \supseteq \langle T_1, v_1 + 1 \rangle$ ) and then it sends a `FILL_IN_PUB_TS` request containing  $ts_{e_1}$  to the following topic manager in the topic order (if any) or directly to the publisher.

Note that,  $TM_{T_1}$  only attaches its local clock to the timestamp for  $e_2$  without incrementing it; thus the two cases can be considered together. Let

us now consider the behaviour of  $s_i$  and  $s_j$  when the notification is triggered by the ENS.

**Subscriber  $s_i$ .** At time  $\tau_n(i, e_1)$ ,  $s_i$  notifies  $e_1$  and then updates its local clock by executing lines 03 - 08 of the notification procedure. In particular,  $s_i$  updates  $sub\_LC_i$  with the pair  $\langle T_1, v_1 \rangle$  (or  $\langle T_1, v_1 + 1 \rangle$ ). At time  $\tau_n(i, e_2)$ ,  $s_i$  is notified by the ENS about  $e_2$ . Since it has updated only the  $sub\_LC_i$  entry corresponding to  $e_1$ , and considering that the value  $v$  has been assigned to  $T_2$  for  $e_2$ , it means that  $sub\_LC_i \leq ts_{e_2}$  and also  $e_2$  can be notified.

**Subscriber  $s_j$ .** At time  $\tau_n(j, e_2)$ ,  $s_j$  notifies  $e_2$  and then updates its local clock by executing lines 03 - 08 of the notification procedure. In particular,  $s_j$  updates  $sub\_LC_i$  with the pairs  $\langle T_1, v_1 \rangle$  (or  $\langle T_1, v_1 + 1 \rangle$ ) and  $\langle T_2, v \rangle$ . At time  $\tau_n(j, e_2)$ ,  $s_j$  is notified by the ENS about  $e_2$ . Looking to  $ts_{e_2}$ , it is not smaller equal than  $sub\_LC_i$  (i.e., there not exists a component where the timestamp is strictly greater than the local clock).

Therefore the condition at line 01 is not satisfied,  $e_1$  is discarded and we have a contradiction.

□*Lemma 2*

**Theorem 1** *Let  $e_k$  and  $e_h$  be two events. If a subscriber  $s_i$  notifies first  $e_k$  and then  $e_h$ , any other subscriber that notifies both  $e_k$  and  $e_h$  will notify  $e_k$  before than  $e_h$ .*

**Proof** The proof trivially follows from Lemma 1 and Lemma 2. □*Theorem 1*

## 4 Causality relation among events

In this Section we show that the events published on the same topic maintain a causality relation among them. To this end, we assume that each process that is publisher on a topic  $T$ , it is also a subscriber of the same topic. Thus, we define a *causality relation* " $\mapsto$ " as follows:

**Definition 5** *If  $publish(e) \mapsto publish(e')$ , where  $e, e'$  are published on the same topic  $T$  and  $e, e'$  have the same destination, then  $e$  has to be notified before  $e'$ .*

Note that this definition meets the original definition of causal ordering given by Birman and Joseph in the context of broadcast communications [6] when considering a publish/subscribe system with a single topic. To demonstrate that in our total ordering algorithm events published on the same topic also respect a causal order, we have to prove that: (i) the relation

“  $\mapsto$  “ follows the *happened-before* relation introduced by Lamport in [14]; (ii) if the relation “  $\mapsto$  “ is verified, then a subscriber that receives  $e, e'$  notifies  $e$  before  $e'$ . To this end, we introduce the following Lemma:

**Lemma 3** *Given two events  $e, e'$  published on a topic  $T$ ,  $\text{publish}(e) \mapsto \text{publish}(e')$  if and only if:*

1.  $e, e'$  are published by the same process, or
2.  $e$  is published by a process  $p$ , while  $e'$  is published by a process  $q \neq p$  after it has notified  $e$ , or
3. there exists  $\text{publish}(e'')$  such that:  $\text{publish}(e) \mapsto \text{publish}(e'') \mapsto \text{publish}(e')$ .

**Proof** For the condition 1, let us assume by absurd that two events  $e, e'$ , published in this order by the same process  $p$ , have the sequence numbers  $sn_e$  and  $sn_{e'}$  such that  $sn_{e'} < sn_e$ . Because the sequence numbers are assigned by the same topic manager  $TM_T$  and we have assumed the presence of FIFO channels,  $TM_T$  will receive  $e, e'$  in the same order they have been published by process  $p$ . Due to the increasing monotonicity of the sequence numbers assigned by the same topic manager, the event  $e'$  cannot have a sequence number lower than the one assigned to  $e$ . This obviously contradicts the thesis.

For condition 2, instead, let us assume by absurd that the event  $e'$  published by a process  $q$  has a sequence number lower than the one assigned to the event  $e$  previously published by a process  $p$ , with  $p \neq q$ . Because these sequence numbers are assigned by the same topic manager  $TM_T$ , and the event  $e$  has been already notified by process  $q$  before publishing  $e'$ ,  $TM_T$  will receive the event  $e'$  after that a sequence number for  $e$  has been assigned. Due to the increasing mononicity of sequence numbers assigned by a topic manager,  $e'$  cannot have a sequence number lower than  $e$ . This clearly leads to a contradiction.

The condition 3 simply follows from conditions 1 and 2 by considering two couples of publish operations, i.e.,  $\text{publish}(e), \text{publish}(e'')$  and  $\text{publish}(e''), \text{publish}(e')$ .  $\square_{\text{Lemma 3}}$

The next step is to demonstrate that if the relation “  $\mapsto$  “ is verified, then a subscriber that receives  $e$  and  $e'$  notifies  $e$  before  $e'$ . To this end, we introduce the following theorem:

**Theorem 2** *If  $\text{publish}(e) \mapsto \text{publish}(e')$ , with  $e, e'$  published on the same topic  $T$ , then  $ts_e < ts_{e'}$  and a subscriber that receives both  $e$  and  $e'$  delivers  $e$  before  $e'$ .*

**Proof** By the design of our algorithm, a process  $p$  subscribed to a topic  $T$  that receives an event published on  $T$ , will also receive a timestamp that will contain all pairs  $\langle T_i, sn_i \rangle$  related to the topics  $T_i$  in the sequencing group of  $T$ . Consider two new events  $e, e'$  published in this order on  $T$ , and let us assume by absurd that the process  $p$  receives the timestamps associated to  $e$  and  $e'$  such that:  $ts_{e'} < ts_e$ .

From Lemma 3, we have that  $sn_e < sn_{e'}$ ; therefore,  $e$  and  $e'$  have to be notified in the same order they are published. By considering the whole set of pairs  $\langle T_i, sn_i \rangle$  contained in the timestamps associated to  $e$  and  $e'$ , we have from Lemma 3 that no two events  $e_i$  and  $e_{i'}$ , that could have been published in this order on a same topic  $T_i$  in between  $publish(e)$  and  $publish(e')$ , have their sequence numbers such that  $sn_{e'_i} < sn_{e_i}$ . As such,  $ts_e < ts_{e'}$ , that contradicts the thesis.  $\square_{Theorem 2}$

## 5 Engineering aspects

**Event Buffering.** The algorithm introduced in Section 3.2 assumes that received events with old timestamps are tagged to indicate that they are notified out-of-order. The main source of out-of-order notifications lies in the fact that two events, possibly published by different publishers, can follow distinct paths through the ENS, before reaching the point where they will be notified to the final recipients. To reduce the number of out-of-order notifications, we can use a buffering strategy on the subscriber side. Every time the `ENSnotify()` primitive returns a new event  $e$ , the algorithm checks through the attached timestamp whether some other event can exist with a smaller timestamp. This check is performed by looking at the sequence numbers included in the timestamp: if the values for all the topics are equal to the corresponding ones stored locally in  $sub\_LC_i$ , except for the topic where the event has been published, that must have a value greater than the local one by one unit, then no event with a smaller timestamp exists that must be still received by the subscriber, and the received event can thus be delivered.

If there is a possibility that an event with a smaller timestamp exists but has not been delivered to the subscriber so far, then the event  $e$  is queued in a buffer able to host a maximum of  $b$  events and a timer for  $e$  is started ( $TTL_e$ ). The event  $e$  is delivered through the `notify()` primitive when one of the following conditions holds: (i) all the events with smaller timestamps have been notified, (ii)  $TTL_e$  expires or (iii) the buffer is full, a new event must be buffered and  $e$  is at the head of the queue.

**Reliability.** Making the algorithm presented so far working reliably in an environment where messages can be lost requires some more minor changes.

The loss of a message during the timestamp generation phase, for example, could lead a publisher to wait forever before publishing an event in the ENS. This problem can be solved with a simple retransmission approach: the publisher periodically re-initiates the procedure for building the timestamp until it receives a correct timestamp for the event. During the timestamp construction procedure,  $TMs$  buffer partially filled-in timestamps and retransmit them as soon as they receive another request for the same timestamp. When a timestamp has been completely filled-in, a message can be routed through the appropriate  $TMs$  to free their buffers. Finally, the internal state of  $TMs$  should be preserved despite possible process failures in order to avoid possible TNO violations. This can be obtained by adopting standard replication techniques.

**Dynamic topic ordering.** The algorithm described so far assumes a fixed topic ordering that is given and known by all the participants. This ordering has a strong impact on the performance of the algorithm at run-time as it is used to decide the content of each timestamp. Depending on the intersection among subscriptions, and on the topic ordering, the timestamp for an event published on a topic can have different sizes spanning from a single entry, up to an entry for every topic in the system. This size impacts the time needed to build the timestamp as it will travel through all  $TMs$  of topic it contains. Ideally, topics where a lot of events are published should thus appear in the highest ranks in the topic ordering such that their timestamp will probably contain less entries. However, accurate statistics on the popularity of topics are not always available at configuration time and, moreover, they only describe statistical properties ignoring transient behaviors that can adversely impact system performance for non negligible time periods. In this Section we describe a topic swapping procedure that modifies the topic ordering adapting it at run-time to the current topic publication popularity.

We assume the presence of a special system topic  $T_s$  subscribed by all  $TMs$ , which is used to advertise that a new topic swapping procedure is happening.  $T_s$  is managed by a  $TM$ , say  $TM_{T_s}$ , as all other topics in the system. In addition,  $T_s$  has an associated sequence number that represents the number of swaps occurred so far in the system, and it is used to clearly define subsequent *ordering epochs*, i.e. periods of time where different topic orderings are considered. All  $TMs$  maintain a local copy of this sequence number in  $LC_{T_s}$ .

The topic swapping procedure relies on a function  $f()$  that, when applied on a topic  $T$ , returns a comparable metric that can be then used by a  $TM$  to check if one of the other  $TMs$  managing topics with lower priorities (i.e. a topic  $T'$  such that  $T \rightarrow T'$ ) in the topic order is a candidate for swapping. In this case a swapping procedure takes place: when  $TM_T$  wants to swap position with  $TM_{T'}$ , it contacts  $TM_{T_s}$  and communicates that  $T$  has to be

exchanged with  $T'$  in the topic order. Then,  $TM_{T_s}$  increments  $LC_{T_s}$  and inserts it in a message together with the new topic order. This message is sent to the  $TM$  with lower priority in the topic order and will traverse all the  $TMs$ ; at the end of the procedure each  $TM$  will be informed about the swap and will update the topic order and the value of  $LC_{T_s}$ . This value determines a new epoch: when a  $TM$  receives a timestamp with a previous sequence number for  $T_s$ , it simply discards that message. The definition of function  $f()$  is tied to the application; from a general point of view, it should take into account the topic publication popularity as this metric can lead to shorter timestamps. However, other aspects can be considered as well. For example  $f()$  could be structured in order to push topics associated to  $TMs$  with more available resources (networking and computational) toward higher priorities where larger loads are incurred.

**Dynamic reordering: correctness proof.** Let us consider two topic managers  $TM_{T_1}$  and  $TM_{T_2}$  respectively of topics  $T_1$  and  $T_2$ , currently in the order  $T_1 \rightarrow T_2$  and with a higher publication rate on  $T_2$ .  $TM_{T_1}$  notices that  $f(T_2) > f(T_1)$ ; thus it contacts  $TM_{T_s}$  for a new swap.  $TM_{T_s}$ , in turn, increments  $LC_{T_s}$  by one and inserts this value in a message containing the new ordering rule  $T_2 \rightarrow T_1$ . The message is forwarded along the ordered sequence including all  $TMs$  and it is returned to  $TM_{T_s}$ . If it does not receive back the message within a timeout expiration,  $TM_{T_s}$  assumes that the message has been lost and resends it. Thus, eventually all  $TMs$  will receive the message and update both the order of  $TMs$  in the chain and the local copy of  $LC_{T_s}$ . All events published in the system have a timestamp containing a sequence number for  $T_s$ : in this way all timestamps received by a subscriber have at least one entry in common and can be *comparable*. When a  $TM$  receives a message with a previous sequence number for  $T_s$ , it simply discards that message. It is worth mentioning that  $T_s$  is not involved in the swapping procedures: it can be always considered as the last topic in the sequence.

During the topic swapping procedure, there is a transitory phase during which some  $TMs$  could not be notified yet about a swap. Consider the topic sequence  $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4 \rightarrow T_5$  and two distinct ordering paths  $T_1 \rightarrow T_2 \rightarrow T_5$  and  $T_1 \rightarrow T_2 \rightarrow T_4$ . At time  $t$ ,  $TM_{T_1}$  contacts  $TM_{T_s}$  for a topic swapping between  $T_1$  and  $T_2$ . Thus,  $TM_{T_s}$  increments  $LC_{T_m}$  and inserts it in a message together with the new order  $T_2 \rightarrow T_1$ . This message will be delivered by  $TM_{T_5}$ ,  $TM_{T_4}$ ,  $TM_{T_3}$ ,  $TM_{T_2}$ ,  $TM_{T_1}$  in this order. Suppose that at time  $t' > t$   $TM_{T_5}$  creates a timestamp for a new publication on  $T_5$ : this timestamp will include  $T_1$  and  $T_2$  other than  $T_5$ . We can distinguish between two cases:

1.  $TM_{T_5}$  did not received the message from  $TM_{T_s}$ , thus the new timestamp contains  $\langle TM_{T_1}, TM_{T_2}, TM_{T_5} \rangle$  in this order. Because at time  $t$

$T_5$  precedes  $T_1$  and  $T_2$  in the chain and  $TM_{T_5}$  did not received the message from  $TM_{T_s}$ ,  $TM_{T_1}$  and  $TM_{T_2}$  did not received that message as well. The execution proceeds seamlessly as before time  $t$  (i.e., before topic swapping).

2.  $TM_{T_5}$  received the message from  $TM_{T_s}$ , thus the new timestamp contains  $\langle TM_{T_2}, TM_{T_1}, TM_{T_5} \rangle$  in this order. Two subcases can verify:
  - $TM_{T_1}$  and  $TM_{T_2}$  received the message from  $TM_{T_s}$ , thus all TMs along this ordering path have updated information about the new topic order.
  - At least one between  $TM_{T_1}$  and  $TM_{T_2}$  did not received the message from  $TM_{T_s}$ , for example  $TM_{T_1}$ . However,  $TM_{T_1}$  can verify that the sequence number for  $T_s$  contained in the timestamp created by  $TM_{T_5}$  has been incremented with respect to its local copy, and from the timestamp itself it can infer the new topic ordering rule. Thus,  $TM_{T_1}$  updates information about  $LC_{T_s}$  and the new topic ordering.

Finally, let us suppose that  $TM_{T_4}$  did not received the message from  $TM_{T_s}$  and creates a timestamp for a new publication on  $T_4$  at a time  $t'' > t'$ , with  $TM_{T_1}$  and  $TM_{T_2}$  that have updated their information based on the timestamp previously created by  $TM_{T_5}$ . When  $TM_{T_1}$  receives the timestamp created by  $TM_{T_4}$ , it notices a previous sequence number for  $T_s$  and, therefore, discards that message.

## 6 Performance Evaluation

In this Section we evaluate the behavior of our ordering module implementing the proposed algorithm. In this evaluation we use SCRIBE as the underlying ENS, and Pastry [20] as a point-to-point communication substrate. We further assume that subscriber and publisher roles are played by the same nodes that constitute the ENS.

We first show how ordering affects the system performance in a large scale environment through a simulation-based study. In such a scenario we also show how it is possible to reduce the impact of ordering through our dynamic topic adaptation based on publication popularity. In addition, we also present how this mechanism is able to adapt publication popularity even when this popularity changes. Then, in a second phase we show the performance of the real prototype we implemented in a small scale; in particular, we provide a comparison between our algorithm and a solution based on JGroups, a toolkit for reliable and ordered multicast communication. This study assesses that with a low/medium number of published events per time

unit the performance of the two algorithms are very close, while for a higher publication rate our solution outperforms the one based on JGroups.

## 6.1 Settings and metrics

We used *FreePastry* [10] to implement and evaluate our ordering algorithm. *FreePastry* is a Java tool that provides both a simulator and a prototype of SCRIBE and Pastry.

In the simulated setting, we considered an underlying physical network characterized by two channels types [2]: *fast channels* for short/medium distance (80% of all links) and *slow channels* for long distance (20% of all links). Both were modeled by a Gaussian distribution with mean latency 21ms and 240ms, and standard deviation 10.85ms and 129.27ms respectively. In the prototype-based setting, we deployed our algorithm on virtual machines hosted in 2.8 GHz quad-core dual processor physical machines interconnected with 10Gbps network links. Each virtual machine was equipped with 1 GB of RAM and Linux Ubuntu 10.4 as the OS. We used the WANem [22] network emulator to emulate links with standard ADSL bandwidth and an average latency of 21ms in order to emulate the behavior of small/medium scale WAN.

The following metrics have been considered:

**End to end latency:** represents the time taken by an event for traveling from the publisher to the last notified subscriber and it is measured in seconds. In our experimental analysis we separately tracked for each event the time needed for building the timestamp and the time taken by SCRIBE to notify all the intended subscribers.

**Percentage of tagged messages:** represents the percentage of messages that the ordering module tags because they have been received out-of-order.

**Percentage of notifications:** represents the ratio between the number of events published and notified to all interested subscribers in a second.

**Bandwidth overhead:** represents the additional bandwidth usage imposed by timestamps and it is measured in bytes. In particular, each timestamp entry  $\langle T_i, sn_i \rangle$  measures 24 bytes: we consider the topic identifier  $T_i$  as a Pastry object identifier (i.e., 16 bytes), and the sequence number  $sn_i$  as a Java *long int* (i.e., 8 bytes).

**Message overhead:** represents the load imposed by our algorithm on topic managers during the construction of timestamps. Specifically, it is the ratio between the number of messages processed by a topic manager to build timestamps for events published on topics managed by other nodes, and the total number of processed messages during the construction of timestamps.

**Percentage of sequence detection:** represents the ratio between the number of event sequences correctly detected by all subscribers and the

number of event sequences occurred during an execution of the algorithm (despite how many subscribers detected them).

All the values reported in the following are the result of at least 10 independent runs (we did not observe standard deviation above 5% of reported values, thus they are not plotted on the curves).

Parameters we vary in our analysis are:

**Event rate:** number of events published per second.

**ENS size:** number of processes in the system.

**Buffer size:** maximum number of messages temporarily stored in the buffer on the subscriber side.

**Number of topics:** number of topics in the system available for subscriptions and publications.

**Number of subscription:** number of topics subscribed by each subscriber.

**Publication model:** we model publications as a probability distribution over the set of topics. We consider random uniform distribution or power-law distribution with shapes 0.269 and 0.901. These two values respectively refer to the 40% and 0.5% of topics having a probability of 80% to be selected for a new publication. Moreover, we consider an additional worst case scenario that consists in publishing always on the last topic in the topic order; this represents a disadvantageous scenario for our algorithm as building timestamps will require messages to travel through a long list of TMs.

**Subscription model:** as for publications, subscriptions are modeled as a probability distribution over the set of topics. Again, we consider random uniform distribution or power-law distribution with shapes 0.269 and 0.901. Moreover, we consider an additional scenario in which subscribers subscribe all topics; this represents a particular scenario where all subscribers are part of a single group where all published events are notified (typical setting for broadcast protocols).

## 6.2 Simulation results

In this Section, we first analyze performance assuming a given static topic ordering and we show how results are strongly influenced by this order, then we switch to a setting where dynamic adaptation is enabled and we show the performance improvements obtainable with the topic swapping procedure. Finally, we also show how the presented ordering algorithm helps in augmenting the percentage of event sequences detected by two different subscribers.

**Static topic ordering.** First, we measure the mean end-to-end latency for event notification by considering both the time spent for timestamp

generation and for event diffusion and notification, varying the ENS size in the range [10-10000]. We consider a scenario with 50 topics subscribed by all subscribers; the event rate was set to 1 event/sec and the simulated time is 30 minutes. The results are reported in Figure 9.

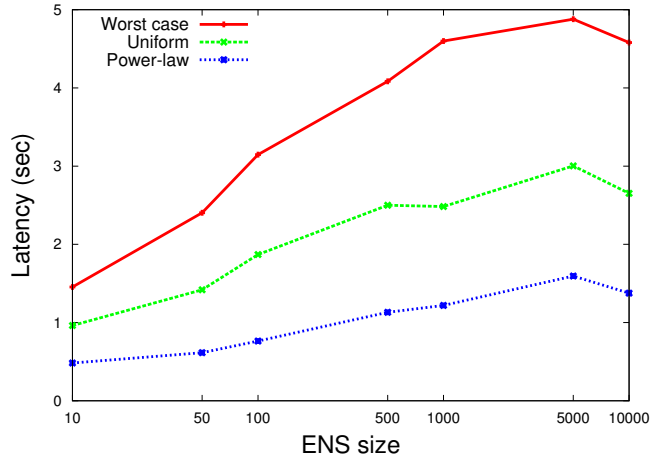


Figure 9: End to end latency for event notification vs ENS size with different publication models.

Each curve refers to a different publication model: worst case, uniform distribution and power-law distribution with shape 0.901. In this last case, the power-law distribution selects more frequently topics at the beginning of the topic order. The curves show the strong impact that different publication models have on notification latency. The coupling between the worst case publication model and the fact that all subscribers subscribe all topics means that the timestamp will travel through all the 50 *TMs* during the generation phase before returning to the publisher and this clearly has a negative effect on the latency that steeply grows with the ENS size. Conversely, in a more favorable scenario where events are published more often on topics with higher priority (power-law model), the latency increment remains reasonable despite the system growth.

In figures 10 and 11 we evaluate separately the time spent for timestamp generation and for event diffusion and notification, considering worst case and power-law publication models respectively. These curves clearly show the impact of the ordering algorithm on latency: it is comparable to event diffusion latency for the power-law model, but it completely drives the overall latency with the worst case model. These curves highlight that our algorithm has a non negligible impact on the event diffusion latency, but this impact can be drastically reduced as long as the topic order is carefully chosen to match the publications popularity.

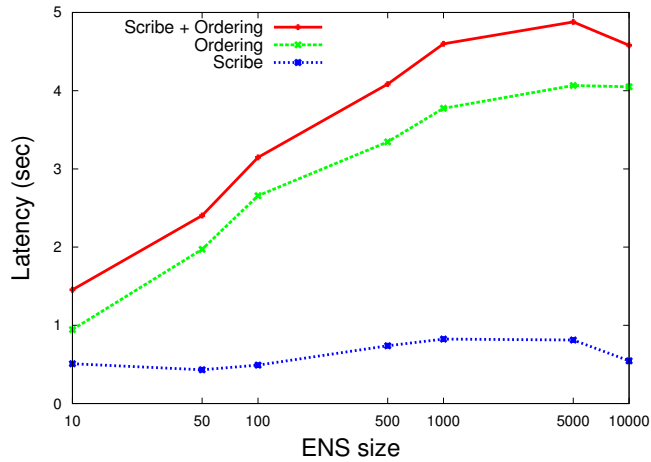


Figure 10: Differentiated ordering and ENS latencies vs ENS size with a worst case publication model.

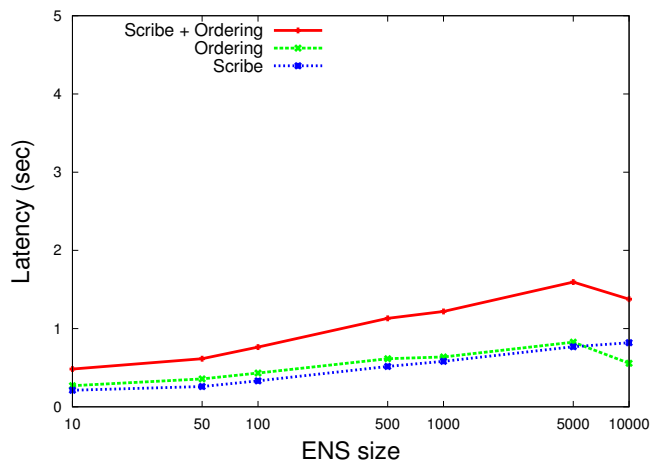


Figure 11: Differentiated ordering and ENS latencies vs ENS size with a power-law (shape 0.901) publication model.

The same consideration can be inferred by looking at Figure 12, that shows the percentage of notifications in a second. Even in this case, an advantageous topic order helps in augmenting the fraction of events notified within a second; this can be particularly useful for high throughput applications.

Figures 13 and 14 illustrate the bandwidth overhead imposed by the usage of timestamps and the message overhead due to the timestamps construction by varying the number of subscriptions. In both cases we consider publication and subscription models based on two power-law distributions

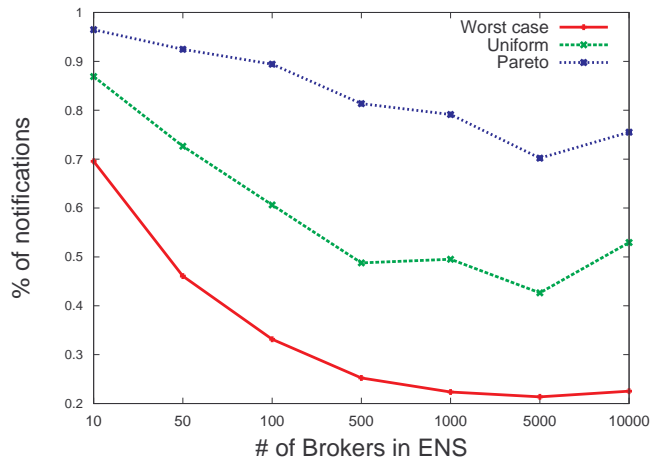


Figure 12: Percentage of notifications in a second with different publication models.

with shape 0.901, one with the most popular topics at the top of the topic order (referred to as *best case*), and one with the most popular topics at the bottom (referred to as *worst case*). In addition, we also consider publication and subscription model based on a uniform distribution.

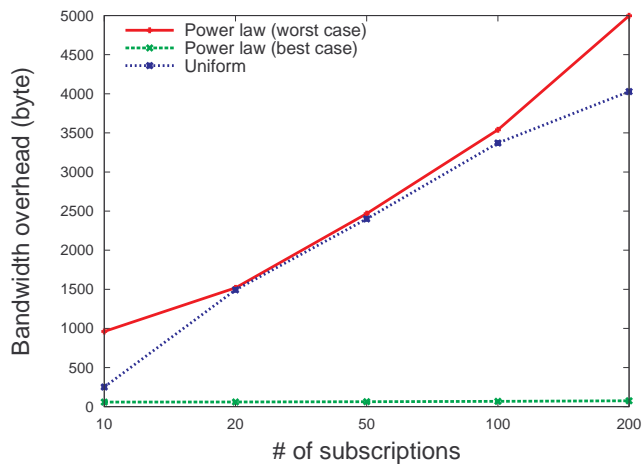


Figure 13: Bandwidth overhead due to the usage of timestamps.

The figures show that when the topic order follows the publication popularity, the overhead imposed by our algorithm both in terms of bandwidth and messages is quite low, and it is not affected by the number of subscription. This is motivated by the fact that in a more *favorable* scenario the timestamp size decreases, and, in turn, the number of processed messages to construct it.

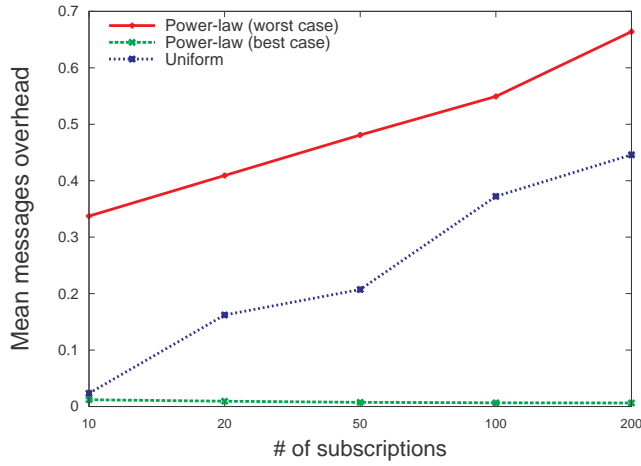


Figure 14: Message overhead imposed by the timestamp construction on topic managers.

Finally, we conclude this part by evaluating the trade-off between the percentage of tagged messages and the delivery latency when we vary the size of the buffer used by subscribers. The simulated scenario is the one described above, with publication and subscription models following a power-law distribution with shape 0.901.

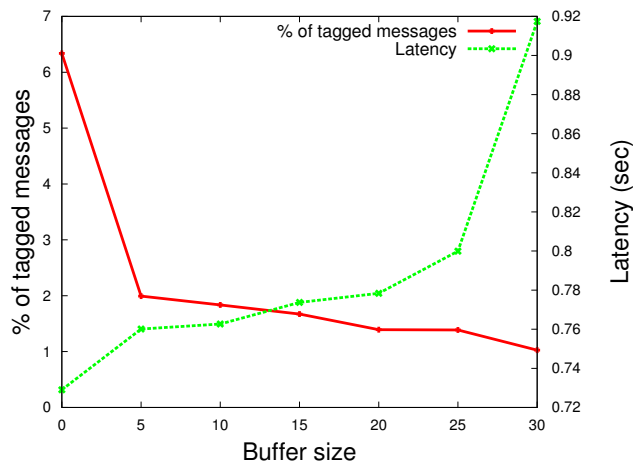


Figure 15: Trade-off between notification reliability and latency vs buffer size.

Figure 15 highlights how the presence of a buffer helps in augmenting the number of ordered messages delivered to the application, at expenses of an increment of the latency. Applications can decide how to tune the system in order to obtain a configuration that satisfies timeliness requirements and/or

helps in delivering a higher number of ordered messages. However, the curves show how just a small buffer can greatly improve this number without impacting too negatively the notification latency.

**Dynamic topic ordering.** In the previous paragraph we have shown the benefit of configuring the topic order in accordance with the topic publication popularity. In this paragraph we evaluate our dynamic topics adaptation algorithm, which aims to adapt at run-time the topics order to publication popularity. The function we adopted in our experiments was  $f(x) = e^{-1/\alpha(x+1)}$  where  $x$  is a sequence number. A topic manager  $TM_{T_i}$  applies this function on the sequence number  $sn_i$  of the topic  $T_i$  it manages and on all sequence numbers of other topic managers in the timestamp it receives: if a sequence number  $sn_j$  exists such that  $f(sn_j) > f(sn_i) + \beta$ , the positions of  $T_i$  and  $T_j$  in the topic order must be swapped. The rationale behind the use of this function is that it eventually converges to  $f(x) = 1$ ; in this way, topics with highest publication rates eventually will reach an almost stable position in the sequence, avoiding swapping procedures that would bring only useless overhead to our algorithm. The parameter  $\alpha$  helps in tuning how fast the function convergence is: a smaller value delays this convergence allowing an higher number of swaps. In this way, the topic sequence quickly adapts to the current publication popularity. However, in order to prevent continuous topics swapping, we allow two topics  $T_i$  and  $T_j$  to swap their position only if  $f(T_j)$  is larger than  $f(T_i)$  for a fixed threshold  $\beta$ . In these tests, we considered a setting with 10000 nodes, 1000 topics, 100 topics subscribed per subscriber and event rate fixed at 1 event/sec. In Figures 16 and 17 show how to tune parameters  $\alpha$  and  $\beta$ .

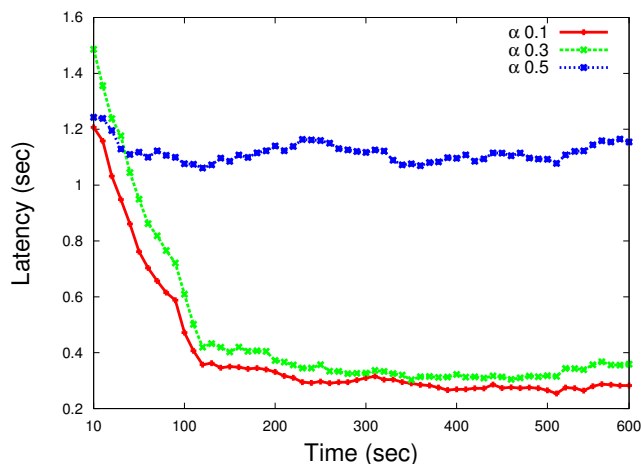


Figure 16: Performance of the algorithm with dynamic adaptation for different  $\alpha$  values.

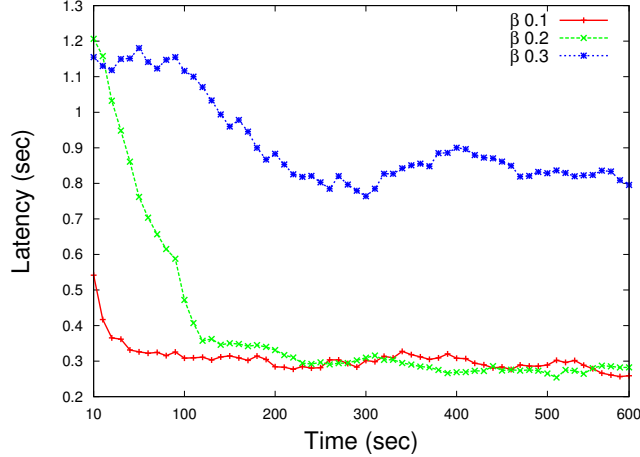


Figure 17: Performance of the algorithm with dynamic adaptation for different  $\beta$  values.

Figure 16 shows the effect of  $\alpha$  on the system, with its value that varies in the set  $\{0.1, 0.3, 0.5\}$ . While the ordering algorithm converges to a small mean latency with  $\alpha = 0.1$  or  $0.3$ , with  $\alpha = 0.5$  there is no benefit from dynamic adaptation: in this case function  $f()$  grows rapidly allowing few topic swaps. The average number of swaps performed during our tests ranged from 15.1 ( $\alpha = 0.1$ ) to 7.4 ( $\alpha = 0.5$ ). Figure 17, instead, shows the effect of  $\beta$  on the system when it varies in the set  $\{0.1, 0.2, 0.3\}$ . With  $\beta = 0.1$  the latency overhead imposed by the ordering algorithm is very low, at the expenses of a high number of swaps (51.8 on average). On the contrary, with  $\beta = 0.3$ , the algorithm convergence speed is lower, due to a reduced number of swaps (7.8 on average). In this case, dynamic topic ordering has no benefit on the performance of our algorithm. The value  $\beta = 0.2$  represents a good trade-off between the two extreme cases: the convergence time of the algorithm to a low latency is still reasonable, with limited overhead imposed by topic swapping. All the following tests were performed setting  $\alpha = 0.1$  and  $\beta = 0.2$ . Figure 18 reports the average end-to-end latency for event notification as the simulation evolves in time.

In this scenario we consider both publications and subscriptions following a power-law distribution with shape 0.901. The *best case* static topic ordering curve assumes that the initial topic order follows the publication popularity distribution. Conversely, the *random* static topic ordering and the dynamic topic ordering curves assume that the initial topic order is randomly chosen. Each point in the picture represents the average notification latency for 10 published events. The curves clearly show how dynamic adaptation allows the ordering algorithm to quickly converge to a small av-

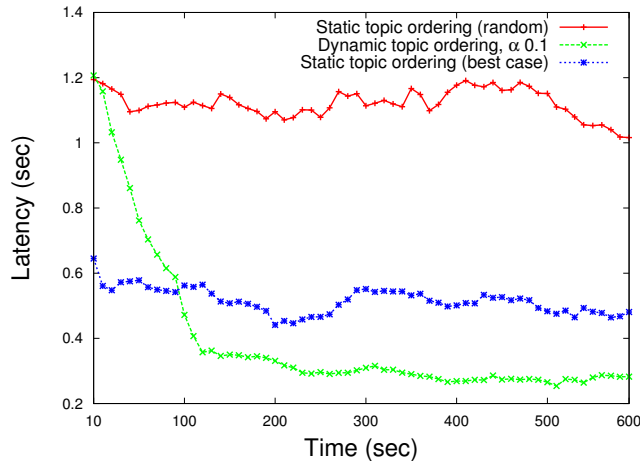


Figure 18: Comparison between the algorithm with dynamic adaptation and the static one (best case, and average case).

average latency even if the starting topic order was random. The interesting aspect is that the dynamic adaptation outperforms the *best case* static ordering: indeed, while topic order in the latter case is decided only on the basis of the statistical properties of the publication popularity distribution, dynamic adaptation is able to tune topic order following the real distribution of publications happening at runtime, thus taking into account also possible temporary fluctuations from the statistical properties of their distribution. In addition, the dynamic topic reordering procedure produces an improvement also in the bandwidth overhead: Figure 19 shows that, with time, due to the reduction of the timestamps size, this overhead is drastically reduced, converging to a mean value about 100 bytes.

Finally, we show the behavior of the ordering algorithm in a special setting where the publication popularity distribution is abruptly changed at runtime.

Figure 20 depicts a scenario in which at time 300 sec. we shuffle the topics list in order to modify the frequency at which topics are returned by the power-law distribution used to model publications. In correspondence of this popularity change, the average latency has a steep increase justified by the fact that the topic order to which the algorithm converged so far is no more the best one for the new publication popularity distribution. However, the dynamic adaptation procedure is able to quickly converge back to a new stable topic ordering that brings back performance in terms of latency to the values shown in the previous tests.

**Impact of notification order on event pattern detection.** In this test we try to recreate a scenario similar to the one that can be encountered in

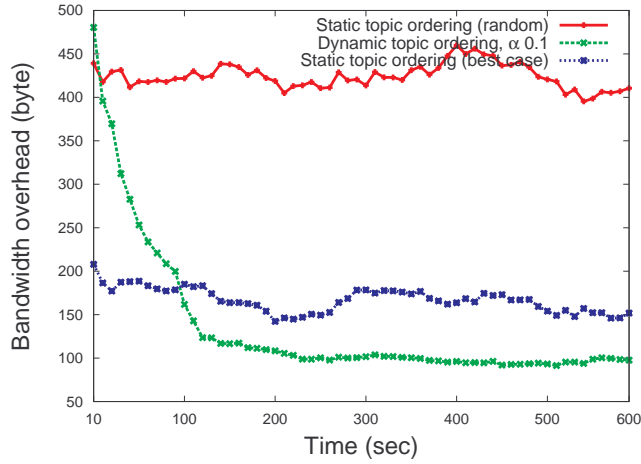


Figure 19: Comparison between the bandwidth overhead in the static and dynamic scenarios.

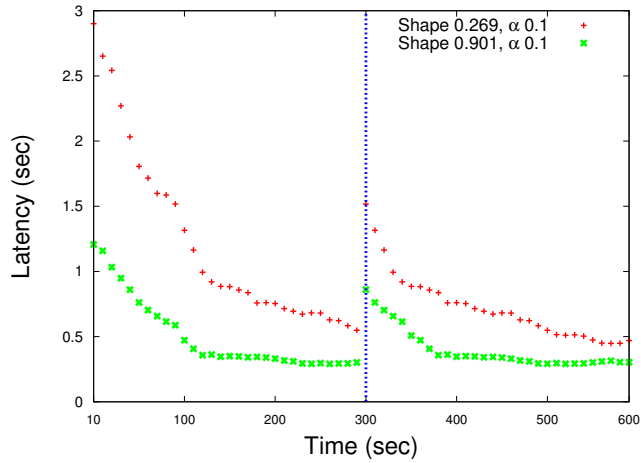


Figure 20: Behaviour of the algorithm with dynamic adaptation with an abrupt change in the system configuration.

composite event pattern detection applications. We consider two subscribers that have to detect the pattern “*event  $e$  precedes  $e'$ , that in turn precedes  $e''$* ”. The system is composed by five publishers that publish on five different topics, two subscribers acting as pattern detectors and an ENS populated with 100 nodes. Each publisher publishes in its topic one of the three events chosen at random at a rate of 5 events/sec. The values reported in Figure 21 show the percentage of sequence detections for different settings. The leftmost bar refers to the bare-bone ENS without ordering: in this case the correct detection of the searched pattern is driven only by chance as the

ENS will not enforce any kind of ordering. As the result shows, only 35% of the patterns are detected by both subscribers.

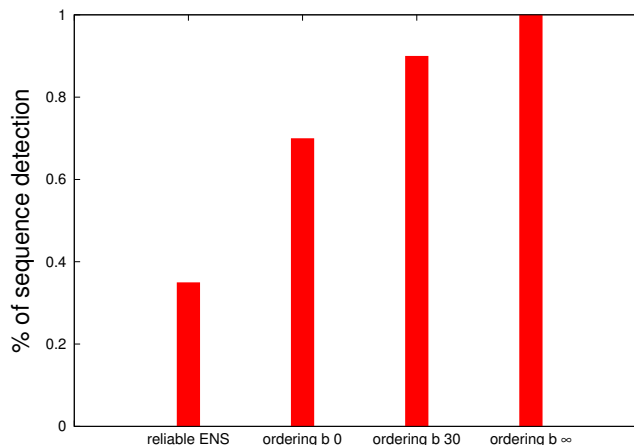


Figure 21: Percentage of consistent discoveries of sequence patterns at two different event engines.

The addition of our algorithm to the game completely changes the figures: the rightmost three bars show the percentage of consistently detected patterns when the algorithm is configured with buffer 0, with buffer 30 and with infinite buffer respectively. In particular, the difference between the bare-bone ENS without ordering and the ENS with the addition of our algorithm, with buffer 0, lies in the fact that the ordering protocol forces events to pass through the network of topic managers, that impose an order among those events. However, several events can be notified out-of-order to different subscribers; hence, the presence of the buffer is not only useful for minimizing out-of-order notifications, but it is also necessary for sequence pattern detection. In fact, note how, in the case with infinite buffer, all the patterns are consistently detected by both subscribers that can thus produce consistent outputs.

**Prototype performance evaluation.** In this Section, we evaluate the performance of our prototype, also comparing the described ordering algorithm with a solution based on the JGroups toolkit. In all our experiments we varied the ENS size from 1 to 10 nodes. In addition, we have a single publisher that can publish on 5 different topics according to the *publication model*, while subscribers are subscribed to all topics. The obtained results are the average of 5 experiments of 5 minutes each.

We first evaluated the *end-to-end notification latency* by varying the *ENS size* and the *publication model*, while the *event rate* is fixed to 1 event/sec.

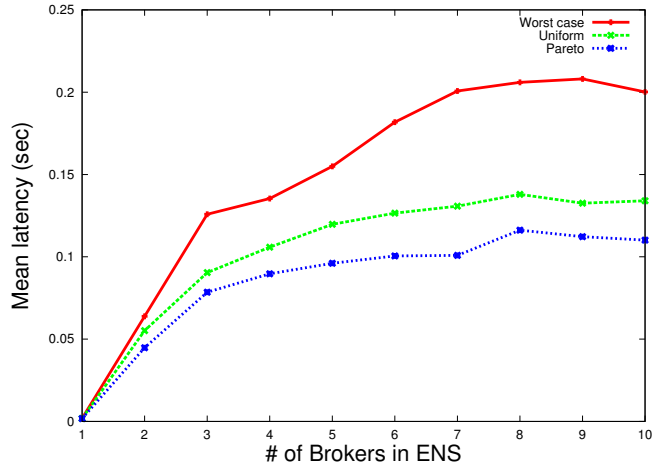


Figure 22: End-to-end notification latency in a real WAN setting.

Figure 22 shows that an increase of the number of nodes in the ENS causes a corresponding increases in the latency. However, the latency quickly reaches a stable point, as its value is mainly driven by the timestamps length. In presence of a single node, the whole system collapses in a centralized sequencer that timestamps and distributes each incoming event.

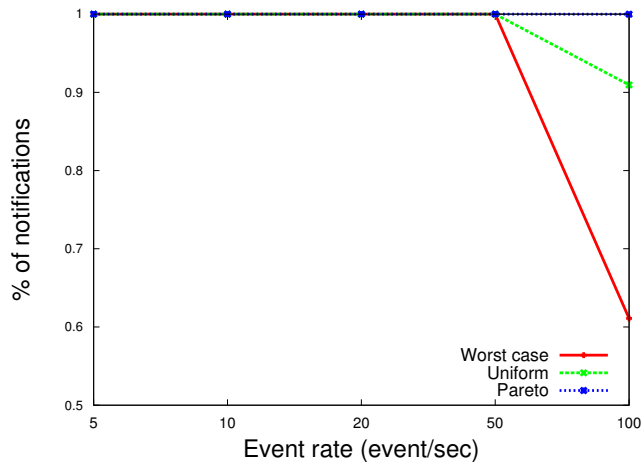


Figure 23: Percentage of notifications in a second by varying the event rate.

In figure 23 we evaluated the *percentage of notifications* by varying the *event rate* in the range [5-100] and the *publication model*, while the *ENS size* is fixed to 10. The figure shows that the overhead imposed by the ordering algorithm in terms of latency may have an impact also on the

percentage of the events notified in a second. This can be particularly disadvantageous in high throughput applications. However, in presence of a more advantageous subscription model, the impact of the latency overhead can be drastically reduced, without affecting the number of notifications per second.

Finally, we provide a comparison between our ordering algorithm and a similar application developed on top of JGroups. Ordered delivery is guaranteed only within a group, thus we developed an application that simply publishes all events in a single group (independently from the topics). Subscribers can then discard upon delivery events published on topics they are not subscribed to.

In our experiment we assumed that no subscriptions or unsubscriptions are issued. JGroups was configured to use the SEQUENCER stack protocol. The usage of the WANem network emulator forced JGroups to use TCP point-to-point channels instead of more performant primitives (e.g. UDP-based IP Multicast). For the ordering algorithm the settings were similar to those described above, with the *ENS* size set to 10 nodes and the *event rate* varying in the range [5 - 100] events/sec.

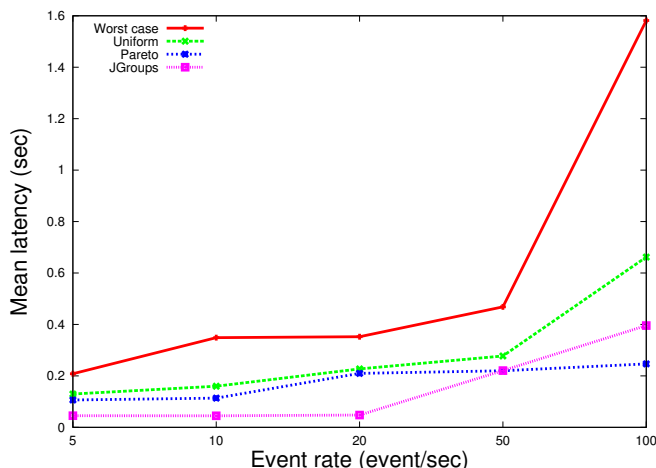


Figure 24: Comparison between our algorithm and a solution based on JGroups. With a high event rate our solution outperforms the one based on JGroups.

Results in figure 24 show how our algorithm delivers performance close to JGroups in terms of mean latency as long as a favorable topic ordering is chosen. When the event rate grows our solution starts to outperform JGroups (see the curve related to the power-law publication model); this is a consequence of two different aspects: while on one side the impact of the ordering algorithm is limited as timestamps are maintained small, on

the other side the application-level tree-based multicast strategy employed by SCRIBE delivers better performance with respect to the point-to-point based multicast strategy employed by JGroups.

## 7 Related work

**Totally ordered communications.** Totally ordered communications have been extensively studied in the literature and there exists a considerable amount of work on total order broadcast primitives following different approaches. A common point is represented by the need of a certain degree of synchronization and knowledge of the system. As an example, [11] is based on a propagation graph to support multiple overlapping groups: authors use a *fixed sequencer* approach in which sequencers are intermediary nodes of the graph placed at the intersection of different groups and messages are propagated through a series of sequencers that order them by merging messages destined to different groups. Differently, Gopal and Toueg in [12] use a *token-based* approach in which the execution of processes is synchronized according to rounds. Lamport in [14] uses a *communication history* approach: messages carry a timestamp and can be broadcasted at any time. Destinations observe the communication history, i.e. previously generated messages and their timestamps, in order to understand when a message must be delivered to preserve total order. Chandra and Toueg [8] use a *destination agreement* approach in which destinations run a consensus algorithm to agree on a set of messages to deliver. Most of the existing total order broadcast approaches and algorithms are extensively surveyed in [9].

All previous algorithms work properly in a small network while they scale badly with respect to number of participants and their geographical distribution. The sequencer represents a bottleneck as well as a single point of failure. Additionally these algorithms require a certain degree of synchronization among the interacting participants and this is in contrast with basic principles of a publish/subscribe paradigm such as time (e.g. asynchronous notifications) and space decoupling. The ordering mechanism proposed in this paper does not require either any prior knowledge on the system or synchronization among the participants but it relies only on the set of available topics and subscriptions, matching thus the publish/subscribe paradigm principles.

An ordering algorithms for publish/subscribe systems is presented in [16]. Similar to our work, authors use a *sequencing network* to order events across multiple groups of subscribers. However, their solution suffers of two problems: (i) it is not able to handle subscription dynamics, and (ii) a new subscription/unsubscription can create loops in the sequencing network (*circular dependency problem*). In this last case, the sequencing network must be rebuilt from scratch. On the contrary, our solution solves these

problems by defining a total order relation among topics that determines a one-way sequence of topic managers that establish an order for events.

Another interesting solution for total ordering in content-based publish/subscribe middleware recently appeared in [24]. Differently from our work, authors define a *Uniform Agreement* property: as such, two correct processes interested in two events  $e$  and  $e'$  both deliver them and they do so in the same order. In [24] each publisher has to *advertise* all brokers about the set of events it will publish. To this end, a broadcast procedure is employed to create a spanning tree rooted at the publisher. Subscriptions are instead forwarded hop-by-hop along the reverse path of matching advertisement trees up to the publisher. While in our solution the correct order of events is reconstructed on subscribers' side, in [24] this task is performed by brokers, that use advertisements and subscriptions to detect a *conflict*, i.e., an out-of-order notification to one or more subscribers. Conflicts are resolved through an acknowledgment mechanism: a broker issues a request to the next hop with conflicting advertisement. The next hop replies back with an ack message if it can determine that there is no conflict. Although an evaluation of a real prototype of the algorithm in a small system shows that it scales well with the number of subscriptions, the broadcast advertisement procedure may pose bandwidth issues in large network. In addition, the acknowledgment mechanism used to avoid out-of-order notifications introduces a feedback among brokers, making this solution not suitable for cloud environments, where one-way communication is preferred [5].

Finally, authors in [23] investigate the problem of multi-delivery multicast in asynchronous systems in presence of crash-stop failures. They introduce an aggregation model based on a predicate grammar for expressing conjunctions of types of events and properties for the multicast primitives. The paper shows that a total order is necessary to guarantee an agreement on events notified to processes interested in identical conjunctions. In particular, this is shown by deploying an algorithm that implements the described aggregation model on top of a total order broadcast and vice-versa for a majority of correct processes.

**Timestamping techniques.** Logical clocks have been introduced by Lamport in [14] to identify the causality relation among events of a distributed computation. In [18], the notion of vector clock has been introduced to capture such causality relation. A vector clock is composed of  $n$  entries, one for each process in the system while a logical clock is an integer. Logical clocks and vector clocks have been used to solve many basic problems such as transaction management, coordination protocols, ordered communication protocols, message stability protocols, distributed predicate detection just to name a few [3].

At a first glance, timestamps used in our algorithm resemble vector clocks, but they are very different structures. Vector clocks have a well

defined and fixed structure that depends on the size of the distributed computation in terms of processes. Therefore the causality relation among two events can be detected just comparing (entry by entry) the two vector clocks associated with the two events. On the contrary, our timestamp structure is independent from the system size but it rather depends on the current set of subscriptions. This is why, the ordering relation among two events can be detected looking, first, to the structure of the timestamps (i.e., the events have to be comparable according to Definition 3) and secondly, if the timestamps are comparable, the ordering among the two events can be detected examining the values contained in common entries of the timestamps. Let us finally remark that in our timestamping technique, the timestamp associated with an event does not bring any information about the producer of the event, this matches the anonymity principle of a publish/subscribe paradigm (producers and consumers do not know each other).

In some work, events are timestamped with physical clocks. As an example, [15] uses accuracy interval-based timestamps relying on NTP synchronized local clocks as a global time reference. The interesting aspect of this timestamping technique lies on the fact that the order of events follows the real time order. However, such a technique has many drawbacks. Many events could be issued in the same physical time interval by producers (so they have the same physical timestamp). A total order could be established among these events only resorting on an additional deterministic information such as the identifier of the publisher of the event, contradicting thus the anonymity principle. Moreover, such protocols can lose liveness during periods in which there is a disconnection with the NTP server.

## 8 Conclusion and future work

Ordering events across topics in a pub-sub system is a complex problem that, if not addressed, can severely impact several kinds of applications that rely on it for their correct functioning. The complexity of this problem lies both in distributed nature of modern ENSs and in the inherent dynamism of publish/subscribe interactions. This paper presented a novel algorithm that can be used to detect and distinguish out-of-order notifications on top of a generic ENS. Our solution improves the current state of the art by providing efficient deterministic out-of-order detection, even in presence of interest changes and without any manual configuration.

However, it is susceptible to further improvements. An aspect that we planned to address in the close future is the failure of topic managers. Currently, we assume the presence of reliable TMs; as such, they are always up and running during the timestamp generation of our protocol. On the contrary, we need to evaluate the impact of a failure of topic managers on the correctness and execution of the algorithm, and how this affects the system

performance. Then, we propose to study a solution that replicates the state of a topic manager on other nodes, in order to improve the reliability of our algorithm.

Another aspect that we planned to study is the impact of dynamic on the ordering algorithm. In particular, we want to evaluate how the change of subscriptions at run-time can impact both the timestamp generation and the dynamic topic reordering procedure. In this Chapter, we described how to manage subscriptions and unsubscriptions in order to preserve the Total Notification Order property, but how they can affect the execution of the algorithm and the topic adaptation to the publication popularity is still unclear.

Finally, a further improvement of our algorithm is to introduce causality relation not only among events published on the same topic, as shown in Section 4, but also among events published on different topics. We are currently planning to work on an extension of our logical timestamping mechanism by using additional physical clock information that allow to notify events in an order that is coherent with the publication time, while both keeping the liveness of the notifications and the anonymity of the publisher/subscriber paradigm.

## References

- [1] R. Baldoni, S. Cimmino, and C. Marchetti. A classification of total order specifications and its application to fixed sequencer-based implementations. *J. Parallel Distrib. Comput.*, 66(1):108–127, 2006.
- [2] R. Baldoni, C. Marchetti, and A. Virgillito. Impact of WAN Channel Behavior on End-to-end Latency of Replication Protocols. In *Proceedings of the Sixth European Dependable Computing Conference*, page 118. IEEE Computer Society, 2006.
- [3] R. Baldoni and M. Raynal. Fundamentals of Distributed Computing: A Practical Tour of Vector Clock Systems. *IEEE Distributed Systems Online*, 3(2), 12 2002.
- [4] A. R. Bharambe, S. Rao, and S. Seshan. Mercury: a scalable publish-subscribe system for internet games. In *Proceedings of the 1st workshop on Network and system support for games*, pages 3–9, 2002.
- [5] K. Birman, G. Chockler, and R. van Renesse. Toward a cloud computing research agenda. *SIGACT News*, 40(2):68–80, 2009.
- [6] K.P. Birman and T.A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems (TOCS)*, 5(1):47–76, 1987.

- [7] M. Castro, P. Druschel, A.M. Kermarrec, and A.I.T. Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. *Selected Areas in Communications, IEEE Journal on*, 20(8):1489–1499, 2002.
- [8] T.D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, 43(2):225–267, 1996.
- [9] X. Defago, A. Schiper, and P. Urban. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys (CSUR)*, 36(4):372–421, 2004.
- [10] FreePastry. <http://www.freepastry.org/>.
- [11] H. Garcia-Molina and A.M. Spaulster. Ordered and reliable multicast communication. *ACM Transactions on Computer Systems (TOCS)*, 9(3):242–271, 1991.
- [12] A.S. Gopal and S. Toueg. Reliable Broadcast in Synchronous and Asynchronous Environments. In *Proceedings of the 3rd International Workshop on Distributed Algorithms*, pages 110–123. Springer-Verlag, 1989.
- [13] JGroups. <http://www.jgroups.org/>.
- [14] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.
- [15] C. Liebig, M. Cilia, and A. Buchmann. Event composition in time-dependent distributed systems. In *Cooperative Information Systems, 1999. CoopIS'99. Proceedings. 1999 IFICIS International Conference on*, pages 70–78. IEEE, 2002.
- [16] C. Lumezanu, N. Spring, and B. Bhattacharjee. Decentralized message ordering for publish/subscribe systems. *Middleware 2006*, pages 162–179, 2006.
- [17] A. Malekpour, A. Carzaniga, G. Toffetti Carughi, and F. Pedone. Probabilistic fifo ordering in publish/subscribe networks. Technical report, Faculty of Informatics, University of Lugano, Technical Report USI-INF-TR-2011-2, 2011.
- [18] F. Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms*, pages 215–226. North-Holland, 1989.
- [19] P.R. Pietzuch, B. Shand, and J. Bacon. Composite event detection as a generic middleware extension. *Network, IEEE*, 18(1):44–55, 2004.

- [20] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware 2001*, pages 329–350. Springer, 2001.
- [21] Real time messaging and integration middleware. <http://www.rti.com/>.
- [22] WANem. <http://wanem.sourceforge.net/>.
- [23] G.A. Wilkin and P. Eugster. Multicast with aggregated deliveries. In *Proceedings of the 1st International Workshop on Algorithms and Models for Distributed Event Processing (AlMoDEP)*. ACM, 2011.
- [24] K. Zhang, V. Muthusamy, and H.A. Jacobsen. Total order in content-based publish/subscribe systems.