Efficient Notification Ordering for Geo-Distributed Pub/Sub Systems

Supplemental material

Roberto Baldoni, Silvia Bonomi, Marco Platania, and Leonardo Querzoni

1 Algorithm Pseudo-code

This section illustrates the pseudocode of the algorithm. Before proceeding with the description, we introduce the local data structures maintained by publishers, subscribers, and topic managers.

Local data structures at 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 in the ENS.

Local data structures at 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 $< T_i, sn_i >$ pairs, 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 $\perp$.
- $to\_deliver_i$: a set variable storing $< e, ts, T >$ triple where e is an event (not in right order) notified by the ENS, ts is the timestamp attached to the event and T is the topic where the event has been published.

Local data structures at each topic manager TM_{T_i}: 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.
- LLC_{T_i}: is a set of $< T_j, sn_j >$ pairs where T_j is a topic identifier and sn_j is a sequence number. Such set contains an entry for each topic $T_j \in \mathcal{SG}_{T_i}$ such that $T_i \rightarrow T_j$.
- $externalSubs_{T_i}$: a set of $< id, sub >$ pairs 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.
- SG_{T_i}: is a set containing identifiers of topics belonging to the sequencing group of T_i.

As an example, let us consider a system where $S_i = \{T1, T2, T3\}$ and $S_j = \{T1, T2\}$ are respectively the two subscriptions of subscribers s_i and s_j. The three variables $externalSubs$ maintained by each topic manager are respectively: $externalSubs_{T1} = \{< i, S_i >, < j, S_j >\}$, $externalSubs_{T2} = \{< i, S_i >, < j, S_j >\}$, and $externalSubs_{T3} = \{< i, S_i >\}$.

The PUBLISH() **Operation.** The algorithm for a PUBLISH() operation is shown in Figure 2. 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 such topic does not exist, then the function returns $null$. If a $null$ value is passed as topic identifier, the function returns the last topic identifier contained in the timestamp.
- getTMAddress(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 $< T, - >$ with the pair $< T, LC_T >$.
- updateLLC($LLC, < T, sn' >$): modifies the set LLC by updating the pair corresponding to topic T with a pair $< T, sn'' >$, where sn'' is the maximum between sn (the sequence number already stored locally in LLC for topic T) and sn' (i.e., the sequence

• *R. Baldoni, S. Bonomi, and L. Querzoni are with the Department of Computer, Control, and Management Engineering, Sapienza University of Rome, Rome, Italy.*
M. Platania is with the Department of Computer Science, Johns Hopkins University, Baltimore MD, USA
E-mail: {baldoni|bonomi|querzoni}@dis.uniroma1.it; platania@cs.jhu.edu

A preliminary and short version of this paper has appeared in the proceedings of the 26th International Parallel & Distributed Processing Symposium.

number contained in the partially filled timestamp).

In addition, we have defined a more complex function, namely updateSequencingGroup($T, externalSubs_T$), that generates the sequencing group $\mathcal{SG}_T$ 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 1.

```
function updateSequencingGroup(T_i, externalSubs):

(01) for each < −, s >∈ externalSubs do SG_{T_i} ← SG_{T_i} ∪ s; endfor
(02) for each T_j ≠ T_i ∈ SG_{T_i} do
(03)     let S = {s ∈ externalSubs | T_j ∈ s};
(04)     if (|S| ≤ 1) then SG_{T_i} ← SG_{T_i}/{T_j} endif
(05) endfor
(06) return SG_T.
```

Fig. 1. The updateSequencingGroup() function (for a topic manager TM_{T_i}).

```
operation PUBLISH(e, T):

(01) id_e ← generateUniqueEventID (e);
(02) outgoingEvents ← outgoingEvents ∪ {(e, id_e, T)};
(03) dest ← getTMAddress(T);
(04) send CREATE_PUB_TS (id_e, i, T) to dest;
-------------------------------------------------
(05) when EVENT_TS (ts, e_id) is delivered:
(06)     let < e, id_e, T >∈ outgoingEvents
(07)     such that (e_id = id_e);
(08)     ENSpublish(< e, ts >, T);
(09)     outgoingEvents ← outgoingEvents/{< e, id_e, T >}.
```

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

```
(01) when CREATE_PUB_TS (e_id, j, T) is delivered:
(02)     for each T_j ∈ SG_{T_i} do ev_ts ← ev_ts ∪ {< T_j, ⊥ >}; endfor
(03)     for each < T_j, sn >∈ LLC_{T_i} do
(04)             ev_ts ← update(ev_ts, < T_j, sn >);
(05)     endFor;
(06)     LC_{T_i} ← LC_{T_i} + 1;
(07)     ev_ts ← update(ev_ts, < T_i, LC_{T_i} >);
(08)     t' ← next(ts, T_i);
(09)     if (t' = null)
(10)        then send EVENT_TS (ev_ts, e_id) to p_j;
(11)             else dest ← getTMAddress(t');
(12)             send FILL_IN_PUB_TS (ev_ts, e_id, j) to dest;
(13)     endif
-------------------------------------------------
(14) when FILL_IN_PUB_TS (ts, e_id, j) is delivered:
(15)     for each < T_j, sn_j >∈ ts such that T_i → T_j do
(16)        LLC_{T_i} ← updateLLC(LLC_{T_i}, < T_j, sn_j >);
(17)     endFor
(18)     ts ← update(ts, < T_i, LC_{T_i} >);
(19)     t' ← next(ts, T_i);
(20)     if (t' = null)
(21)        then send EVENT_TS (ts, e_id) to p_j;
(22)        else dest ← getTMAddress(t');
(23)             send FILL_IN_PUB_TS (ts, e_id, j) to dest;
(24)     endif.
```

(b) TM Protocol (for a topic manager TM_{T_i} belonging to the multistage sequencer of T)

Fig. 2. The publish() protocol.

When an event e is published in a topic T, the publisher p_i executes the algorithm shown in Figure 2(a). In particular, it associates with 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 with T (lines 03-04).

Receiving the CREATE_PUB_TS (id_e, i, T) message, the topic manager TM_T executes the algorithm shown in Figure 2(b). In particular, it first creates an empty timestamp ts_e containing an entry $< T_j, \perp >$ for each topic T_j belonging to the sequencing group of T (line 02), it updates the sequence numbers of topics T_k following T in the topic ranking with the values locally stored in LLC_T (line 04), it increments its local sequence number and updates the corresponding entry in ts_e (lines 06 - 07). Finally, TM_T 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. The publisher (Figure 2(a)) pulls the event from the buffer, attaches the timestamp and publishes both on the ENS (lines 06 - 09). Note that, when a topic manager receives a FILL_IN_PUB_TS message, it just attaches its local sequence number (line 18) and update its local LLC variable to keep track of the sequence numbers associated to topics with lower rank (lines 15 - 17).

The NOTIFY() **Operation.** When an event e is notified by the ENS, a subscriber s_i executes the algorithm shown in Figure 3. To simplify the pseudo-code of the algorithm, we defined the following basic functions:

- isNext(ts, LC): The function takes as parameter a timestamp ts and a local subscription clock LC and returns a boolean value. In particular, let k be the size of the timestamp, the function returns true if and only if there exist $k-1$ $< T_i, sn_i >$ pairs in ts equal to those stored in LC and the last $< T_j, sn_j >$ pair in ts is such that $< T_j, sn' >\in LC$ and $sn_j = sn' + 1$.
- updateSubLC($sub\_LC, < T, sn' >$): modifies the set sub_LC by updating the pair corresponding to topic T with the pair $< T, sn'' >$ where sn'' is the maximum between sn (the sequence number already stored locally in sub_LC for topic T) and sn' (i.e., the sequence number contained in the timestamp).

A subscriber s_i first checks if the event e is a subscriptionUpdate event for some topic T (line 01). If so, the subscriber checks if all previous events published on its subscribed topics have been notified by comparing the subscription timestamp with the local subscription one. This is done by checking entry by entry that all the sequence numbers (except the one associated to T) stored in the local subscription timestamp are equal to the one contained in the event timestamp -1. In this case, s_i uses the event timestamp to update its local subscription clock sub_LC_i (lines 02 - 05), otherwise it buffers the event and processes it later.

If the event is a generic application event, s_i checks if the topic T of the notified event is actually subscribed and then checks if it has been notified by the ENS in the right order (line 10). If such condition is satisfied, the event e is notified to the application (line 11) and the local subscription clock sub_LC_i is updated with

```
upon ENSnotify(< e, ts >, T):
(01)  if (e = subscriptionUpdate)
(02)    then if (∀ < T_j, v >∈ sub_LC_i | T_j ≠ T, ∃ < T_j, v + 1 >∈ ts)
(03)           then for each < T_j, sn_j >∈ ts such that (T_j ∈ subs_i) do
(04)                  updateSubLC(sub_LC_i, < T_j, sn_j >);
(05)                endFor
(06)           else to_deliver_i ← to_deliver_i ∪ {< e, ts, T >};
(07)         endif
(08)  endif
(09)  if ((T ∈ subs_i) ∧ (e ≠ subscriptionUpdate))
(10)    then if (isNext(ts, sub_LC_i) = true)
(11)           then trigger notify (e, T);
(12)                for each (< T_j, v >∈ sub_LC_i)
(13)                  if(∃ < T_j, v' >∈ ts)
(14)                    then updateSubLC(sub_LC_i, < T_j, v >);
(15)                  endif
(16)                endfor
(17)           else to_deliver_i ← to_deliver_i ∪ {< e, ts, T >};
(18)         endif
(19)  endif
      ---------------------------------------------------------------
(20)  when ∃ < e, ts, T >∈ to_deliver_i|isNext(ts, sub_LC_i) = true
(21)    to_deliver_i ← to_deliver_i \ {< e, ts, T >};
(22)    trigger notify (e, T)
(23)    for each (< T_j, v >∈ sub_LC_i)
(24)      if (∃ < T_j, − >∈ ts)
(25)        then updateSubLC(sub_LC_i, < T_j, v >);
(26)    endFor
```

Fig. 3. The notify() protocol (for subscriber s_i).

```
operation SUBSCRIBE(T):
(01)  ENSsubscribe(T);
      ---------------------------------------------------------------
(02) upon ENSsubscribeReturn(T):
(03)  subs_i ← subs_i ∪ {T};
(04)  for each T_j ∈ subs_i do ts ← ts ∪ {T_j, ⊥} endFor
(05)  t' ← next(ts, null);
(06)  dest ← getTMAddress(t');
(07)  send FILL_IN_SUB_TS (ts, (subs_i ∪ {T}), id) to dest;
      ---------------------------------------------------------------
(08)  when COMPLETED_SUB_VC (ts, s) is delivered:
(09)    sub_LC_i ← ts;
(10)    e ← subscriptionUpdate;
(11)    for each T_j ∈ subs_i do ENSpublish(< e, ts >, T_j) endFor;
(12)    trigger subscribeReturn(T);
```

(a) Subscriber Protocol

```
(01) when FILL_IN_SUB_TS (ts, sub, j) is delivered:
(02)  externalSubs_i ← update (externalSubs_i, < j, sub >);
(03)  SG_{T_i} ← updateSequencingGroup(T_i, externalSubs_{T_i});
(04)  for each < T_j, sn_j >∈ ts such that ((T_i → T_j) ∧ (T_j ∈ SG_{T_i})) do
(05)        LLC_i ← LLC_i ∪ {< T_j, sn_j >};
(06)  endFor
(07)  LC_{T_i} ← LC_{T_i} + 1
(08)  ts ← update (ts, < T_i, LC_{T_i} >);
(09)  t' ← next (ts, T_i);
(10)  if (t' = null)
(11)    then send COMPLETED_SUB_VC (ts, s) to j;
(12)    else dest ← getTMAddress(t');
(13)         send FILL_IN_SUB_TS (ts, s, j) to dest;
(14)  endif
```

(b) TM Protocol

Fig. 4. The subscribe() protocol.

the sequence numbers contained in the event timestamp (lines 12 - 16).

On the contrary, if the event e is not in the right order, the subscriber s_i buffers e (line 17) and continuously checks, by comparing its timestamp with the local subscription clock, when it is in the right order (lines 20 - 26).

The SUBSCRIBE() **and** UNSUBSCRIBE() **Operations.** The algorithm for a SUBSCRIBE() operation is shown in Figure 4. To simplify the pseudo-code of the algorithm, 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 $< T, sn >$ pairs, 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 4(a). In particular, it first invokes the ENSsubsribe(T) operation to make the subscription active on the ENS (line 01) and blocks until such operation completes. When the subscription is active on the ENS, s_i adds the topic T to the list of subscribed topics and creates an empty subscription timestamp through the createSubTimestamp function (including also topic T). 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 03-07).

Upon the delivery of a FILL_IN_SUB_TS message, each topic manager $TM_{T'}$ executes the algorithm shown in Figure 4(b). In particular, $TM_{T'}$ updates its $externalSubs_k$ variable with the new subscription (line 02), recomputes the sequencing group $\mathcal{SG}_{T'}$ (line 03) and updates the sequence numbers of topics with lower rank (lines 04 - 06), increments its local sequence number (line 07), updates its entry in the subscription timestamp (line 08), and finally forwards the FILL_IN_SUB_TS message to the preceding topic manager until it is completed and returned to the subscriber.

When the subscriber s_i receives the completed subscription timestamp, it updates its local subscription clock (line 09) and publishes a subscriptionUpdate event on each topic T_j belonging to its subscription to keep other subscribers aligned with the subscription local clock (lines 10 - 11). Finally, it returns from the operation to notify the application that the subscription is now active (line 12).

The algorithm for the UNSUBSCRIBE() operation is shown in Figure 5. 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 5(b)), topic managers update the $externalSubs$ set accordingly with the received subscription, recompute the sequencing group, and remove

```
operation UNSUBSCRIBE(T, subID):
(01) subs_i ← subs_i/{T};
(02) for each T_j ∈ subs_i do
(03)     dest ← getTMAddress(T_j);
(04)     send UPDATE_SUB (subID, subs_i) to dest;
(05) endfor
(06) dest ← getTMAddress(T);
(07) send UPDATE_SUB (subID, ∅) to dest;
(08) ENSunsubscribe(T);
```

(a) Subscriber Protocol

```
when UPDATE_SUB (id, s) is delivered:
(01) if (s ≠ ∅)
(02)     then externalSubs_i ← update(externalSubs_i, < id, s >);
(03)     else externalSubs_i ← externalSubs_i/{< id, − >};
(04) endif;
(05) SG_{T_i} ← updateSequencingGroup(T_i, externalSubs_{T_i}).
(06) for each < T_j, − >∈ LLC_{T_i} such that (T_j ∉ SG_{T_i}) do
(07)         LLC_{T_i} ← LLC_{T_i} \ {< T_j, − >};
(08) endFor.
```

(b) TM Protocol

Fig. 5. The unsubscribe() protocol.

topics that are no more in the sequencing group from the LLC_T set.

2 CORRECTNESS PROOF

In this section we will show that the TNO property holds for any pair of events.

Definition 1: Given a generic subscriber s_i, let us denote $\tau_n(i, e)$ the time instant in which s_i is notified of e by the system.

Lemma 1: Let e_1 and e_2 be two events both published in a topic T. If a subscriber s_i notifies e_1 before e_2 and the sequencing group $\mathcal{SG}_T$ does not change during the two publish operations, 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, and that 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_x notifying both e_1 and e_2, it follows that at time $\tau_n(x, e_1)$, $T_1 \in S_x$ and at time $\tau_n(x, e_2)$, $T_2 \in S_x$. Moreover, $sub\_LC_x \leq ts_{e_1}$ at time $\tau_n(x, e_1)$ and $sub\_LC_x() \leq ts_{e_2}$ at time $\tau_n(x, e_2)$.

Given the two timestamps ts_{e_1} and ts_{e_2} associated respectively with 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 2(a) and it sends a CREATE_PUB_TS message. Let us assume, without loss of generality, 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 LC_T at 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}, adds to ts_{e_1} the pairs $< T_j, sn_j >$ stored locally in LLC_T and containing sequence numbers associated with all the topics T_j preceding T in $\mathcal{SG}_T$ according with the topic rank, increments its local clock (i.e., $LC_T = v + 1$), and includes the pair $< T, v + 1 >$ in ts_{e_1} (lines 03 - 04, Figure 2(b)). Two cases can happen:

1) ts_{e_1} **contains only the entry for** T **and for topics in** LLC_T **(line 05):** $ts_{e_1} = \{< T, v+1 >, < T_j, x > \cdots < T_j, y >\}$ (with $< T_j, x > \cdots < T_j, y > \in LLC_T$) and TM_T returns the completed timestamp to the publisher for the publication in the ENS.
2) ts_{e_1} **contains more entries (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 $< T', \perp >$ contained in ts_{e_1} with the current value of $LC_{T'}$ and 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_1} to the publisher (lines 11 - 16).

When TM_T delivers the CREATE_PUB_TS message sent by p_h, it follows the same steps: it creates a template ts_{e_2} for the timestamp, increments its local sequence number (i.e., $LC_T = v + 2$), includes the pair $< T, v + 2 >$ in ts_{e_2}, and sends the timestamp to the publisher or to the following topic manager.

Let us note that TM_T updates sequence numbers of topics stored in LLC_T by following a monotonic increasing order, i.e., it always takes the maximum between the one already stored locally and the one in the received timestamp (cfr. lines 15-17, Figure 2(b)).

Considering that the sequencing groups are not changing, 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 with each topic. In particular, considering that (i) a topic manager can only increment its local sequence number when a publication occurs, and (ii) topic managers are connected through FIFO channels, it follows that for each topic T_i the sequence number v' associated with T_i in ts_{e_2} cannot be smaller than the one associated with T_i in ts_e. 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 (line 14, Figure 3), and that (ii) $ts_{e_1} < ts_{e_2}$, we have that s_j evaluating the notification condition at line 09 will store the event e_2 in the $to\_deliver_j$ buffer. This leads to a contradiction as e_2 will never be notified before e_1.

$\square_{Lemma\ 1}$

Lemma 2: Let e_1 and e_2 be two events published respectively in topics T_1 and T_2, with $T_1 \neq T_2$. Let us

assume that the sequencing groups $\mathcal{SG}_{T_1}$ and $\mathcal{SG}_{T_2}$ do not change during the two publish operations. 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 only two topics subscribed by both s_i and s_j[1] (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 in 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_x, if it notifies both e_1 and e_2, it follows that, at time $\tau_n(x, e_1)$, $T_1 \in S_x$ and at time $\tau_n(x, e_2)$, $T_2 \in S_x$. Moreover, $sub\_LC_x \leq ts_{e_1}$ at time $\tau_n(x, e_1)$ and $sub\_LC_x \leq ts_{e_2}$ at time $\tau_n(x, e_2)$.

Given the timestamps ts_{e_1} and ts_{e_2} associated respectively with 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 precedence than T_2 in the topic order.

Considering how sequencing groups (and consequently timestamps) are defined by the updateSequencingGroup function shown in Figure 1, each event published in T_1 will have attached a timestamp containing the pairs $< T_1, x_1 >, < T_2, x_2 >$ and each event published in T_2 will have attached a timestamp containing the pairs $< T_1, y_1 >, < T_2, y_2 >$.

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 2(a)). Receiving such a request, TM_{T_2} executes line 02 of Figure 2(b) and creates an empty event timestamp containing entries for T_1 and T_2 (i.e., $ts_{e_2} \supseteq < T_1, \perp >, < T_2, \perp >$), increments its local clock, let's say to a value v (line 06), updates its component of the timestamp with its local clock (i.e., $ts_{e_2} \supseteq < T_1, \perp >, < T_2, v >$), and 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}). Delivering such message, TM_{T_1} will execute lines 15 - 17, Figure 2(b) by storing locally the pair $< T_2, v >$ in its LLC_1 variable.

The same procedure is executed when e_1 is published. Note that, since $T_1 \rightarrow T_2$, it follows that the pair $< T_2, v >$ contained in ts_{e_2} will be attached to ts_{e_1}.

In the worst case scenario, due to concurrency in the timestamp creation procedure, TM_{T_1} can either deliver first the CREATE_PUB_TS message sent from the publisher of e_2 and then the FILL_IN_PUB_TS message sent by TM_{T_2} or vice-versa, it can first manage the FILL_IN_PUB_TS message and then the CREATE_PUB_TS one:

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., $ts_{e_1} \supseteq < T_1, \perp >, < T_2, \perp >$), updates the entry related to T_2 with the pair $< T_2, v' >$ stored locally in LLC_1 (with $v' \leq v$), updates its local clock to $v_1 + 1$, updates its timestamp component with its local clock (i.e., $ts_{e_1} \supseteq < T_1, v_1 + 1 >, < T_2, v' >$), and 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 2(b), and updates its timestamp component with its local clock (i.e., $ts_{e_2} \supseteq < T_1, v_1 + 1 >, < T_2, v >$). 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 FILL_IN_PUB_TS message for ts_{e_2}, TM_{T_1} executes line 18 of Figure 2(b), updates its timestamp component with its local clock (i.e., $ts_{e_2} \supseteq < T_1, v_1 >, < T_2, v >$), and updates its LLC_1 variable by storing the pair $< T_2, v >$. 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., $ts_{e_1} \supseteq < T_1, \perp >$), updates the entry related to T_2 with the pair $< T_2, v >$ stored locally in LLC_1, updates its local clock to $v_1 + 1$, updates its timestamp component with its local clock (i.e., $ts_{e_1} \supseteq < T_1, v_1 + 1 >$), and 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.

Let us now consider the behavior 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 04 - 11 of the notification procedure. In particular, s_i updates sub_LC_i with the pair $< T_1, v_1 >$ (or $< T_1, v_1 + 1 >$). 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 receives e_2 that contains the entry $< T_1, v_1 + 1 >$ associated to e_1. However, since e_1 has not yet been received, the sub_LC_i data structure is not updated and the isNext() function returns false. As a consequence, s_j will buffer e_2 for a future notification when e_1 will

1. 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.

be notified, and we have a contradiction.

$\square_{Lemma\ 2}$

Lemma 3: Let s_i be a subscriber that invokes a subscribe(T) operation at time t. If the ENS is reliable, then s_i eventually generates the subscribeReturn(T) event.

Proof The subscribeReturn(T) event is triggered by a subscriber s_i in line 12, Figure 4(a) when it delivers a COMPLETED_SUB_VC message. Such message is generated by the topic manger TM_{T_k} responsible of the highest ranked topic in the subscription of s_i (line 11, Figure 4(b)) when delivering a FILL_IN_SUB_TS message. Such message is originally generated by the subscriber itself and then forwarded by topic managers responsible of topics in the subscription (line 13, Figure 4(b)). Considering that (i) the FILL_IN_SUB_TS message is generated by the subscriber after the subscription is active on the ENS, (ii) the ENS guarantees that eventually such event happens, and (iii) messages are not lost in the forwarding chain, we have that the claim simply follows.

$\square_{Lemma\ 3}$

Lemma 4: Let s_i be a subscriber that invokes a subscribe(T) operation at time t and let $t+\Delta$ be the time at which the subscribe operation terminates. If the ENS is reliable, then s_i will notify all the events published in T at time $t' > t$.

Proof Let us suppose by contradiction that there exists an event e published in the new subscribed topic T after the subscription operation ends and that s_i never notifies such event. When the subscription terminates, the subscriber s_i updates its local subscription clock with the pairs contained in the subscription timestamp (line 09, Figure 4(a)). Let $sub\_LC_i = \{< T_i, v_i >, < T, v > \cdots < T_k, v_k >\}$ be such local subscription clock at time $t + \Delta$. Let us consider now the first event e published after time $t + \Delta$. When e is published, the publisher executes the algorithm in Figure 2 requesting topic managers in its sequencing group to fill in its timestamp. In particular, TM_T creates the timestamp and fills in its entry by incrementing its sequence number and adding the pair $< T, v+1 >$. When such event is notified to s_i, it checks if the event can be notified immediately as it is the next with respect to the local subscription timestamp. Two cases can happen:

1) isNext(ts, sub_LC_i) = true. In this case, the event is immediately notified and we have a contradiction.
2) isNext(ts, sub_LC_i) = false. In this case, the event is stored in the $to\_deliver_i$ buffer while the subscriber waits until this events will be the next to be notified. Let un note that the ENS is reliable. Thus, published events will be eventually notified and the local subscription clock will be increased until the condition becomes true and the claim follows.

$\square_{Lemma\ 4}$

Lemma 5: Let $\mathcal{SG}_{T_1}, \mathcal{SG}_{T_2}, \cdots .\mathcal{SG}_{T_n}$ be the sequencing groups of topics $T_1, T_2, \ldots T_n$ at time t. Let s_i be a subscriber that invokes subscribe(T) at time t and let $\mathcal{SG}'_{T_1}$, $\mathcal{SG}'_{T_2}$, $\ldots \mathcal{SG}'_{T_n}$ be the sequencing groups after the subscription operation ends at time $t+\Delta$ (i.e., when s_i triggers subcribeReturn(T)). For each T_i, $\mathcal{SG}_{T_i} \subseteq \mathcal{SG}'_{T_i}$.

Proof For each topic T_i, its sequencing group $\mathcal{SG}_{T_i}$ is calculated by considering the union of all the subscriptions containing T_i, stored locally at TM_{T_i} in the $externalSubs_i$ variable, and will include all the topics T_j such that there exist at least two subscriptions including both T_i and T_j (lines 02- 05, Figure 1).

At time t, when a subscriber s_i invokes a subscribe(T) operation, it executes the protocol in Figure 4(a) and will advertise that its subscription is changed by sending a FILL_IN_SUB_TS message that will flow through all the TMs responsible for topics in the subscription, from the last to the first in the topic order.

When a topic manager TM_j delivers such FILL_IN_SUB_TS message at a certain time $t' \in [t, t+\Delta]$, it will update its $externalSubs_j$ set with the new subscription (line 02, Figure 4(b)) and will use this set to compute the new sequencing group $\mathcal{SG}'_{T_j}$ when creating timestamps.

For each of these topics T_j, updating the $externalSubs_j$ variable with a subscription including T may cause a change of $\mathcal{SG}_{T_j}$ and the following cases can happen:

1) $\mathbf{T \in \mathcal{SG}_{T_j}}$ **at time t**. If $T \in \mathcal{SG}_{T_j}$, it means that $T \in externalSubs_j$ at time t. Thus, the new subscription of s_i has no effect on it, $\mathcal{SG}_{T_j} = \mathcal{SG}'_{T_j}$ and the claim follows.
2) $\mathbf{T \notin \mathcal{SG}_{T_j}}$ **at time t**. Two further cases can happen:
   a) $\mathbf{T \notin externalSubs_j}$ **at time t**. At time t, T was not included in $externalSubs_j$, meaning that there were no subscriptions containing together both T and T_j. As a consequence, the new subscription will be the only one including both topics. As a consequence, T will be not considered in the computation of the new sequencing group for T_j (lines 02- 05, Figure 1), $\mathcal{SG}_{T_j} = \mathcal{SG}'_{T_j}$ and the claim follows.
   b) $\mathbf{T \in externalSubs_j}$ **at time t**. In this case, since $T \notin \mathcal{SG}_{T_j}$ at time t, it means that there exists at most another subscription including both T and T_j. As a consequence, the addition of the new subscription containing T to $externalSubs_j$ creates a pair of subscription sharing at least two same topics. The protocol will include T in the new sequencing group (lines 02- 05, Figure 1). Note that the insertion of a topic in $externalSubs$ can not remove any other topic already part of the sequencing

group. As a consequence, in this case we will have that $\mathcal{SG}_{T_j} \subset \mathcal{SG}'_{T_j}$ and the claim follows.

$\square_{Lemma\ 5}$

Lemma 6: Let s_i be a subscriber that invokes a subscribe(T) operation at time t and let $\mathcal{SG}'_T$ be the sequencing group of T after the subscription. If s_i notifies an event e published on the topic T, then the timestamp for e has been built according to $\mathcal{SG}'_T$.

Proof Subscriber s_i triggers the ENSsubscribe() only after it has inserted the topic T in its subscription (lines 07-08, Figure 4(a)) and has updated its local clock sub_LC_i. In particular, this happens when s_i delivers a COMPLETED_SUB_VC (ts, s) message (line 05, Figure 4(a)). The received subscription timestamp ts contains an entry for each T in the subscription of s_i. In particular, each entry is filled in with the current local clock LC_i incremented by one (line 03, Figure 4(b)) when TM_{T_i} delivers a FILL_IN_SUB_TS and updates its current sequencing group.

Let us show in the following that for any event e published on the topic T and notified to s_i, its timestamp will be generated following the new $\mathcal{SG}'_T$.

If e has been notified by s_i, then each entry of its local clock sub_LCi is smaller or equal than the corresponding entry in the timestamp ts_e of e, and it is strictly smaller for at least one entry. Considering that filling in ts_e each TM copies (or increments and copies) the entry in the timestamp, it follows that every TM has first filled in the subscription timestamp and then ts_e. Considering that (i) the subscription takes effect at each TM just after FILL_IN_SUB_TS message is delivered, and (ii) this message induces a change on the sequencing group, it follows that ts_e is created and filled in according to $\mathcal{SG}'_T$ and the claim follows.

$\square_{Lemma\ 6}$

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 Lemmas 1 - 6 and considering that when a subscriber s_i invokes an unsubscribe(T) operation, TNO violations cannot happens because T is immediately removed from the subscription (line 01, Figure 5(a)) and the notification condition becomes false (line 01, Figure 3).

$\square_{Theorem\ 1}$