

## Total Order in Content-Based Publish/Subscribe Systems<sup>‡</sup>

Kaiwen Zhang\*, Vinod Muthusamy<sup>†</sup>, Hans-Arno Jacobsen<sup>†</sup>

*Middleware Systems Research Group*

*Department of \*Computer Science, <sup>†</sup>Electrical and Computer Engineering  
University of Toronto*

*kzhang@cs.toronto.edu, {vinod, jacobsen}@eecg.toronto.edu*

**Abstract**—Total ordering is a messaging guarantee increasingly required of content-based pub/sub systems, which are traditionally focused on performance. The main challenge is the uniform ordering of streams of publications from multiple publishers within an overlay broker network to be delivered to multiple subscribers. Our solution integrates total ordering into the pub/sub logic instead of offloading it as an external service. We show that our solution is fully distributed and relies only on local broker knowledge and overlay links. We can identify and isolate specific publications and subscribers where synchronization is required: the overhead is therefore contained to the affected subscribers. Our solution remains safe under the presence of failure, where we show total order to be impossible to maintain. Our experiments demonstrate that our solution scales with the number of subscriptions and has limited overhead for the non-conflicting cases. A holistic comparison with group communication systems is offered to evaluate their relative scalability.

### I. INTRODUCTION

The pub/sub paradigm [1], [2], [3], [4], [5], [6] provides a simple communication metaphor for entities that require complex interaction patterns while remaining decoupled. Data producers *publish* data to a *broker* which forwards the data to consumers who have *subscribed* to it. Producers and consumers thus remain anonymous. Applications in this space include business process execution [7], workflow management [2], business activity monitoring [8], stock-market monitoring [9], selective information dissemination and RSS filtering [3], complex event processing for algorithmic trading [10], and network monitoring and management [8].

Distributed content-based pub/sub systems are a class of pub/sub designs in which subscribers can express fine-grained constraints over the content of publications, and an overlay network of brokers collaborates to evaluate these subscriptions and delivers matching publications to interested subscribers [1], [6], [11], [12].

Current distributed content-based pub/sub systems do not natively offer strong guarantees about the order in which publications are delivered to subscribers. For example, due to network delays, it is possible that two subscribers interested in the same set of published events receive them in different order. This paper develops a total order algorithm to ensure

that all recipients of the same set of publications will deliver those publications in an uniform order.

There are many pub/sub applications that could benefit from total order guarantees. For example, in a distributed on-line game ([13], [14]), players receive publications to notify them of changes to the game state, such as the movement of another player, or the points and objects accumulated by a player. In this case, users receiving messages in different order may perceive inconsistent game states. Similarly, in electronic stock tickers, investors are notified of the trades and price updates on a set of stocks [9]. It would be unfair for investors to observe different market behavior due to differences in the order they receive stock updates. Another set of examples are workflow or business process engines in which the flow of the process is driven by a carefully orchestrated sequence of trigger messages [15]. It is common in these systems to monitor the messages received by elements of the workflow, perhaps to detect an event pattern that reveals a missed business opportunity. Ordering guarantees here are critical, as the event pattern is sensitive to the order in which publications are delivered. Another case for total order arises in situations where a user's program state needs to be replayed. For example, if the exact sequence of updates received by an air traffic controller's console could be mirrored by a replica, it can be used by auditors to determine whether the controller's decisions were justified based on his view of the system. On a related note, developers could monitor the trace of publications received by a software component using an external subscriber to replay a sequence of events and debug faults in the component.

Broadly speaking, there are at least two classes of applications that require some kind of total order guarantee. First are those that need to detect event patterns or composite events [5], such as detecting stock market trends, or observing attack signatures. The others are those where an application's state is constructed as a function of the sequence of events it receives, that is, those that follow the Command [16] or Event-sourcing [17] design patterns. An example of this includes the online gaming scenario above.

Offering total order guarantees in a distributed content-based pub/sub system is a challenging problem. First, we note that the naïve solution of a centralized sequencer to globally order messages does not scale, violates the

<sup>‡</sup>We would like to acknowledge Dr. Hein Meling for revising the paper.

distributed nature of the system, and is overkill since it is only necessary to order those messages that are delivered to multiple subscribers.

Existing distributed messaging systems that offer various ordering semantics are not readily applicable to our context. The primary reason is that existing systems typically assume messages are delivered to an explicit and well-formed group of nodes. For example, consensus algorithms assume all nodes participating in the protocol are aware of everyone in the group [18]. The same is true of group communication middleware [19], and channel-based messaging systems [20]. As well, distributed algorithms based on logical clocks assume a relatively stable and small group of participants [21].

In content-based pub/sub, there are no explicit groups; each subscription may overlap the interests of another, and every publication may be delivered to any subset of subscriptions. Consequently, given  $n$  subscriptions there are potentially  $2^n$  groups, and it quickly becomes infeasible to manage an exponentially increasing number of groups.

In this paper, we develop a distributed protocol to ensure total ordering of publications in a content-based pub/sub system. The protocol works in two parts: first during the *detection* phase, a broker determines whether a publication needs to be ordered or if it can be immediately forwarded. Next, for those publications that require ordering, in the *resolution* phase the broker collaborates with a small number of other brokers in the overlay to determine a consistent delivery order.

The protocol is encapsulated within the broker network, and the algorithm does not change the system model of popular pub/sub implementations [1], [11], [6]. In particular, the network is assumed to be asynchronous, and the brokers are decoupled and have knowledge only of their overlay neighbors.

The paper makes the following key contributions:

- 1) Sec. III describes how a weaker ordering semantic called per-publisher order can be supported using FIFO links between nodes. We also discuss the impossibility of total ordering in crash failure scenarios.
- 2) Sec. IV develops a distributed total ordering algorithm in a content-based pub/sub model. The algorithm is fully distributed, and requires no changes to the pub/sub client API. The basis of the algorithm rests on FIFO links which can partially support total order. We also show the safety of our algorithm when failures are introduced to the model.
- 3) Sec. V evaluates the performance of an implementation of the algorithm. The results show that the pairwise total order algorithm scales with the number of subscriptions in the system and the publication size overhead is bounded. We also conduct a holistic comparison with group communication systems.

The paper continues with a discussion of related work in Sec. II. Proofs to theorems included in this paper can be

found in our technical report<sup>1</sup>.

## II. BACKGROUND AND RELATED WORK

The problem of total order multicast has been widely studied in the literature [22]. Within the context of pub/sub, a publication message  $m$  is multicast to a set of subscribers  $Dest(m)$  (destination group). In that respect, the usual definitions of validity, agreement, and integrity remain the same. On the other hand, total order semantics can be defined in multiple ways, each as a stronger version of the previous one:

- (Per-Publisher Total Order) For any two messages  $m$  and  $m'$  and any two processes  $p$  and  $q$ , where  $\{p, q\} \in Dest(m) \cap Dest(m')$ . If  $sender(m) = sender(m')$ , then  $p$  delivers  $m$  before  $m'$  if and only if  $q$  delivers  $m$  before  $m'$ .

Essentially, every message sent by a certain publisher will be delivered in same order to all its recipients. This order can be enforced by the publisher alone (e.g., through FIFO ordering). A stronger property considers the destination groups only [22]:

- (Local Total Order) For any two messages  $m$  and  $m'$  and any two processes  $p$  and  $q$ , where  $\{p, q\} \subseteq Dest(m) = Dest(m')$ .  $p$  delivers  $m$  before  $m'$  if and only if  $q$  delivers  $m$  before  $m'$ .

In this case, there must be some sort of ordering enforced to all messages of a group, regardless of the publisher. Finally, we abstract away the concept of destination group and only consider messages received by common receivers [22]:

- (Pairwise Total Order) For any two messages  $m$  and  $m'$  and any two processes  $p$  and  $q$ . If  $\{p, q\} \subseteq Dest(m)$  and  $\{p, q\} \subseteq Dest(m')$ , then  $p$  delivers  $m$  before  $m'$  if and only if  $q$  delivers  $m$  before  $m'$ .

Sec. III shows how per-publisher total ordering can be enforced using FIFO links. We then extend the solution in Sec. IV to maintain pairwise total order.

Ordering semantics can also be qualified as *weak* if they do not guarantee reliability. We define the properties of liveness and safety within the context of failures as follows:

- (Liveness) If a message  $m$  is sent by a process  $p$  to its destination group  $Dest(m)$ , then all processes in  $Dest(m)$  will eventually deliver  $m$ .
- (Safety) If messages  $m$  and  $n$  are delivered to both processes  $p$  and  $q$ , then  $m$  and  $n$  are delivered to both  $p$  and  $q$  in the appropriate order.

A weak total ordering semantic maintains *safety* but not *liveness*. In other words, processes are not required to deliver every message they receive as long as they maintain the right order amongst delivered messages [23].

Fig. 1 illustrates the various types of total order. In this example, we have three publishers (A-C) and six subscribers

<sup>1</sup> Available here: <http://msrg.org/papers/total-order-tr>

(1-6). Publications by A are delivered to the group  $G_x$ , while B and C publish to the  $G_y$ . Some subscribers are part of both groups (Subscribers 1 and 5). Per-publisher total order is enforced (note the subscribers' ordering need not correspond with the sender's order). Local total order is not respected since the delivery order of  $G_y$  publications is not uniform (because of Subscriber 3). Thus, pairwise total order is not respected. Also note that the delivery order between the triangle and the hexagon at subscribers 1 and 5 violates pairwise total order, but not local total order.

It is interesting to note that in a topic-based environment, local total order is easier to guarantee than pairwise total order. The subscription matrix, that is, the relationships among publishers and subscribers, is well defined and allows sequencing to be separated by topic. On the other hand, pairwise total order requires synchronization between messages of different topics.

In a content-based environment, enforcing local total order strictly requires knowledge of the subscribers set for any given message. Messages are ordered only when two messages have identical delivery groups. Since subscriptions to a message are computed dynamically, the system must first compute the whole set of subscribers per message before making a decision. For pairwise total order, it suffices to show that the messages are *doubly-overlapping* [24]: they have at least two subscribers in common, regardless of the actual composition of the groups.

**Existing Solutions** — There has been considerable work in developing total order broadcast and multicast algorithms [22]. One technique is to route all messages to one or more dedicated sequencers to order messages [25], [26], [27]. A concern with these algorithms is the scalability of the approach as the sequencer(s) must have the capacity to route and process all messages in the system. By contrast, the solution in this paper is fully distributed and uses ad-hoc techniques to resolve ordering conflicts.

Another class of algorithms record the history of interactions among a group of nodes, and nodes use this history to deterministically compute a message order [21], [28]. The solutions in this space assume a well-defined group of nodes that participate in the protocol. Such a model is not appropriate for a distributed content-based pub/sub system. First, in such systems, group membership knowledge is distributed and there can be constant churn in the set

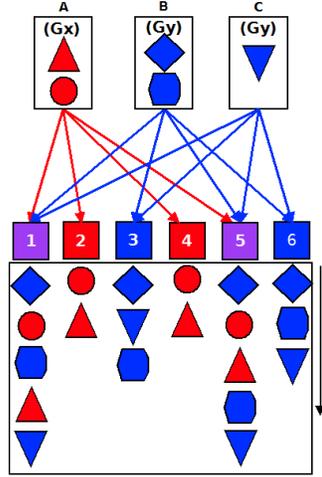


Figure 1. Comparison of Total Ordering

of subscribers and subscriptions. Moreover, subscriptions can express fine-grained interest—as opposed to a broad interest in all messages sent to a group—resulting in an exponential number of groups. For anything but small subscription populations, it is not feasible to form groups. The algorithm in this paper locally determines a set of subscriptions and advertisements that need to participate in resolving the order of a given publication. By performing this detection *on-demand* for every publication, and using only *local* knowledge, the problem becomes tractable.

Previous work on ordering in pub/sub systems focuses on topic-based systems only and uses sequencer nodes to enforce an order in ambiguous situations [24]. The drawback of sequencer nodes is that the traffic is rerouted to be delivered from the sequencer, independent of the source of the publication. This limits the advantage of using overlay broker networks since it can create a bottleneck near those sequencers. Furthermore, global knowledge of the subscriptions is required for the sequencers.

FAIDECS offers a fair decentralized solution for event correlation in event-processing systems [29]. The solution relies on a DHT lookup to identify *merger* processes which are responsible for ordering specific subsets of event types. Publications must flow through this network of mergers before being delivered. Because the solution makes no assumption on the underlying model, all events must be processed through FAIDECS. In contrast, our solution is integrated directly as a lightweight component installed on pub/sub brokers, and can leverage local broker knowledge about the underlying overlay topology to limit the ordering overhead.

The solution in this paper orders publications by synchronizing brokers along the path of publications. It does not require additional knowledge from the brokers. Furthermore, it is a solution for content-based systems.

In [30], FIFO links are used to preserve an ordering enforced by merger nodes. Our solution differs in that it requires no modification to the topology or dedicated nodes for ordering. We also leverage FIFO links as the foundation of enforcing a certain order in our system. Brokers then detect situations where FIFO links are not sufficient to maintain total order and run a resolution protocol to reorder the stream.

### III. ANALYSIS OF ORDERING SEMANTICS

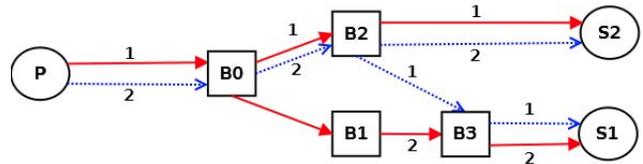


Figure 2. FIFO Links Counter-Example

**Per-Publisher Total Order in Content-Based Publish/Subscribe Systems** — In a failure-free content-based model with FIFO links, per-publisher total order is provided

natively as long as the delivery path from the publisher to a given subscriber is the same for all messages destined to that subscriber (see Theorem 1). The order the messages are sent by a publisher will be maintained as long as the messages are processed and forwarded in the received order by each broker. The client can then safely deliver the messages as they are received in total order.

**Theorem 1.** *Per-publisher total order is maintained using FIFO links.*

The requirement of uniform delivery path per publisher is necessary for cyclic topologies. In such instances, FIFO links are not sufficient to maintain total order. Consider Fig. 2, where the dashed (blue) message is sent before the continuous (red) message; they should therefore be delivered in the same order at both subscribers. However, the continuous message takes a different path from the dashed message to reach  $S1$ . Thus, there is no ordering guarantee between the messages sent by  $B1$  and  $B2$  to  $B3$ .  $S1$  can therefore receive the messages out-of-order.

**Impossibility of total order in publish/subscribe** — Pairwise total order is impossible in publish/subscribe systems under the presence of failures. The proof<sup>2</sup> is a derivation from the same result for total order multicast in asynchronous systems with crash failures [31]. In other words, maintaining both safety and liveness is not possible. As a consequence, we focus our attention on the safety of our algorithm when dealing with failures by maintaining *weak total order*. However, it is possible to design a system where liveness is prioritized over safety. For instance, a timeout mechanism can be used to control the time allocated to the total order component. If total order is not achieved within this time, the message is forwarded immediately to the recipients, ensuring progress.

#### IV. TOTAL ORDER ALGORITHM

We present an algorithm for maintaining pairwise total order in content-based publish/subscribe systems. The algorithm is integrated within the broker logic and does not require any external processes. Scalability is achieved by enforcing and ordering only when necessary: FIFO links are sufficient to preserve total order when certain conditions are met. If not, potential ordering conflicts are detected using local broker knowledge and resolved amongst a small subset of relevant brokers.

**System Model** — We assume an advertisement-based pub/sub protocol that employs reverse path forwarding [1]. There is no restriction on the publication data model and the subscription language; we only assume it is possible to match publications against subscriptions, and find intersections between advertisements and subscriptions.

<sup>2</sup>All proofs are available in the tech report: <http://msrg.org/papers/total-order-tr>

We further assume an acyclic overlay topology of brokers with FIFO links between brokers. We believe this is a reasonable assumption since the underlying messaging can support it (i.e., TCP) and it can also easily be implemented at the pub/sub level by sequencing messages at both ends of a link. Under this model, per-publisher total ordering is provided natively as described in the previous section.

In order to support the required pub/sub protocol, brokers maintain a list of advertisements and subscriptions received at the broker. Brokers are also equipped with a routing table which maintains next-hop information for advertisements and subscriptions.

Our system guarantees weak pairwise total order under the presence of failure. Link failures are prevented using reliable messaging (i.e., TCP). Node failures can result in undeliverable or missing publications, but not in out-of-order messages.

The above properties of the system model are common in a number of distributed content-based pub/sub system [1], [11], [6].

**Ordering conflicts** — We define a conflict as an out-of-order delivery of publications to one or more subscribers. We wish to detect any possible conflicts and resolve them prior to delivery, making our solution pessimistic in nature. The use of FIFO links provides a “natural” ordering for many situations.

**Theorem 2.** (*Natural Total Order*) *Given publications  $p$  and  $q$ , which are delivered to both  $s_1$  and  $s_2$ ,  $p$  is delivered using paths  $P_1$  and  $P_2$  to  $s_1$  and  $s_2$ , respectively. Similarly  $q$  is delivered using paths  $Q_1$  and  $Q_2$  to  $s_1$  and  $s_2$ , respectively. If  $P_1 \cap P_2 \cap Q_1 \cap Q_2 \neq \emptyset$ , then FIFO links preserve pairwise total ordering between  $p$  and  $q$  for  $s_1, s_2$ .*

According to Theorem 2, it is sufficient for publishers to share a common broker along the path to all overlapping subscribers to guarantee total ordering using FIFO links. Fig. 3 illustrates this property: Broker B3 is common to all paths from publishers to S1, S2, S3 and will enforce some ordering on the publications which will be retained by FIFO links.

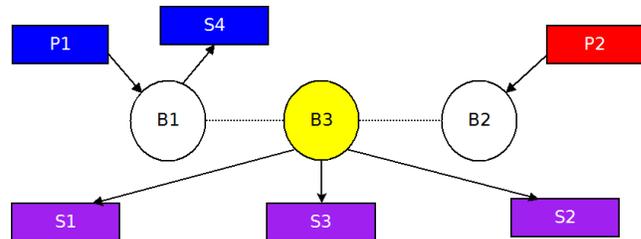


Figure 3. Natural Total Order using FIFO links

Conflicts can therefore only arise when the overlapping subscribers do not have a common broker (see Fig. 4). In this example, the path from P1 to S1 (P1-B1-S1) does not share a common node with the path from P2 to S2 (P2-

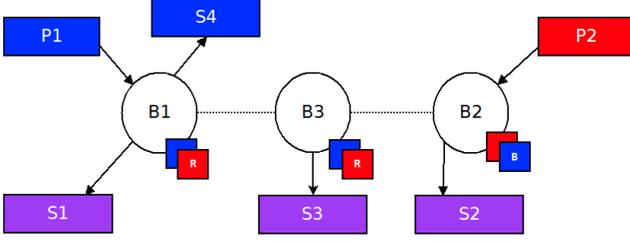


Figure 4. Conflict Example

B2-S2). If P1 and P2 publish in parallel, B1 will receive the publication B from P1 before publication R from P2. Similarly for B2, publication R from P2 are received before P1's B. Since FIFO order is preserved, S1 will then receive the publications in the order enforced by B1 while S2 uses the ordering by B2, where those orders are not guaranteed to be uniform. To resolve this conflict, three solutions are possible:

- 1) Mobility and topology reconfiguration: Natural total ordering can be restored by migrating peers to different brokers. Brokers can be reorganized into a different topology to alter the delivery paths.
- 2) Adaptive routing: Publications can be routed differently to include a common broker on the delivery path.
- 3) Conflict resolution: A conflict resolution process is engaged when a conflict is detected to enforce an ordering on the publications.

We will focus on the third solution, as it imposes the least demand on the system. The two other solutions are left as future work.

**Conflict Detection** — We now present a conflict detection algorithm for brokers. The algorithm requires no global knowledge: any additional state tracked by the algorithm can be derived or obtained from the usual process of advertisement-based forwarding [1].

The algorithm is triggered upon receipt of a publication and can only detect *potential* conflicts based on advertisements. A broker can not instantly determine whether there exists another publication in transit conflicting with this one. It can however determine if there is a possibility of a conflicting publication coming from a certain publisher. If this possibility is detected, the conflict resolution algorithm is triggered (Theorem 3). In other words, the algorithm determines, based on the local knowledge of the broker at the moment the publication is received, whether or not to trigger conflict resolution.

**Theorem 3.** *Given a publication  $p$ , Algorithm 1 detects all potential ordering conflicts with  $p$ .*

Given a publication  $p$ , the algorithm, executed at every broker along the delivery tree, must specifically detect the following in order:

- 1) A set of hops  $H$  and  $|H| > 1$ , where  $h \in H$  if there exists a subscriber that matches  $p$  whose next hop is

$h$ ,

- 2) At least one advertisement  $a$  whose next hop  $h_a$  belongs to  $H$ ,
- 3) And  $a$  matches the intersection of two different subscriptions  $s \cap t$ , where the next hop  $h_a = h_s$  and  $h_a \neq h_t$  such that  $s$  and  $t$  both match  $p$ .

Point 1 is necessary to detect that all subscribers do not share the same next hop. If they do, then the current broker constitutes a common broker for the delivery of all publications upstream of this broker. Any potential conflicts occurring downstream will be processed by the next hop brokers. Thus, the current broker does not require further processing.

Point 2 ensures that there are potentially conflicting publishers. If all the publishers do not share any next hops with subscribers, then the current broker is again common to delivery to all subscribers, *regardless* of the next hops of the subscribers.

Point 3 verifies that any publisher downstream can issue a conflicting publication. In particular, we detect whether it is possible for such a publisher to create a publication  $q$  which will be delivered to an overlapping subscriber  $s_1$  without passing through the current broker *and* have this publication delivered to another subscriber  $s_2$  through the current broker. This would indicate that the delivery path from  $p$  to  $s_2$  and  $q$  to  $s_1$  would be disjoint, in which case a conflict can occur. If the publisher only matches subscribers at the same next hop from the current broker, then the next hop broker can detect and resolve any conflict.

---

**Algorithm 1:** conflictDetection( $p$ , in)

---

```

1 if |next(p)| = 1 then
2   forward(p, h), h ∈ next(p);
3 else
4   foreach h ∈ next(p) do
5     foreach s ∈ match(p), where nh(s) = h do
6       foreach a ∈ pubs(h, s) do
7         if
8           X ≠ ∅, where X = nhs(a) ∩ next(p) \ {h, in}
9         then
10          foreach t ∈ match(p), nh(t) ∈ X where
11            isMatching(a, s ∩ t) do
12              conflictResolution(p, h, nh(t), s);
13              find another matching t or skip loop to
14              next h;

```

---

Algorithm 1 uses two hash tables: the  $pubs(h, s)$  table is indexed by subscribers, returning the list of advertisements matching  $s$ , where  $a$  must have next hop  $h$ ; the  $nhs(a)$  table is indexed by advertisements and returns list of next hops for subscribers matching  $a$ . These data structures are computed during advertisements and (un)subscriptions.

The function  $match(p)$  returns matching subscribers for  $p$  on outgoing links and is provided by the broker logic. The function  $next(p)$  computes the set of next hops for those matching subscribers.  $forward(p, h)$  forwards the publication down the link  $h$ . The forwarding code checks if  $p$  is modified in any way for that specific hop  $h$ . This is

necessary for the conflict resolution explained in the next section.  $conflictResolution(p, h, g, t)$  is a special conflict resolution protocol for forwarding a publication down to a hop  $h$ , knowing there is a conflict with hop  $g$  due to subscription  $t$ .  $nh(s)$  is the next hop for a subscriber  $s$ .  $isMatching(s_1, s_2)$  determines whether two interest spaces are intersecting or not.

The input for the algorithm is the publication  $p$  and  $in$ , which is the incoming edge.

**Conflict Resolution** — Once a conflict is detected, the broker must defer delivery to subscribers until it is resolved. If there are no conflicting publications in transit at the time the conflict is detected, the broker can safely deliver the publication. Otherwise, the broker must deliver both conflicting publications in the correct order. This order is enforced using publication IDs. Each publication’s ID is a combination of its publisher’s ID and a sequence counter which is monotonically increasing for that publisher. The publishers’ IDs are unique and provided by the pub/sub system.

The resolution is processed through acknowledgments. The broker will piggyback a request on the publication to next hops with conflicting advertisements. The next hop brokers process those requests and send an acknowledgment back if it can determine that there is no conflict. Otherwise, the request continues to be carried downstream.

Note that the conflict resolution process does not prevent a broker from deferring forwarding to brokers. If a conflict is detected and the next hop is a broker, the publication is still forwarded, but tagged as a conflict. The next broker processes it as usual, but understands that it must wait for an acknowledgment from the incoming edge before delivering to subscribers who match the specified advertisements from the incoming edge. This is formalized in Theorem 4.

**Theorem 4.** *Given a publication  $p$ , which has been detected to have a conflict with advertisement  $a$  (with possible publication  $q$ ) at broker  $b$ , the resolution algorithm maintains total order.*

---

**Algorithm 2:**  $conflictResolution(p, h, g, t)$

---

```

1 flagR(p, f) ;
2 flagW(p, h) ;
3 addSub(p, t) ;
4 append f to acks(p, h) ;
5 if h is a subscriber then
6 | delay(h) ;

```

---

Algorithm 2 maintains a table  $acks(p, h)$  of pending acknowledgments necessary before delivering  $p$  to  $h$ .  $flagR(p, h)$  and  $flagW(p, h)$  flags  $p$  with a special request or wait flag for that hop, respectively.  $addSub(p, t)$  stores a subscription which conflicts with the publication. The arguments  $(p, h, g, t)$  correspond to the publication  $p$ , the hop  $h$  to deliver and the hop  $g$  which is conflicting and its subscription  $t$ .  $delay(h)$  delays all subsequent publications

to  $h$  until  $deliver(h)$  is called.

We now integrate the conflict resolution mechanism with the detection.  $brokerForwarding$  (Algorithm 3) is invoked upon receiving a publication  $p$  from incoming edge  $in$ .  $isFlagR(p)$  and  $isFlagW(p)$  are used to determine if the publications carry a certain flag.  $safeDetection(p, in)$  verifies if a conflict is possible and/or resolved and flags the publication appropriately.  $returnAck(p, h)$  returns an acknowledgment for  $p$  on hop  $h$ .  $subsC(p)$  returns all subscriptions added to the publication through  $addSub(p, t)$ . Note that the publication does not retain the flags from the incoming publication; they must be set again through  $conflictDetection()$  by this broker. Forwarding only occurs at the end of the algorithm; publications are buffered until then.

---

**Algorithm 3:**  $brokerForwarding(p, in)$

---

```

1 if isFlagW(p) then
2 | flagW(p, h) ;
3 | append in to acks(p, h) for h = nh(c), ∀c ∈ subsC(p) ;
4 conflictDetection(p, in) ;
5 if isFlagR(p) then
6 | safeDetection(p, in) ;
7 | if acks(p, in) = ∅ then returnAck(p, in) ;
8 | else
9 | | append acks(p) to acks(p, h) for all h ≠ in, where h is a
9 | | | subscriber
10 commitForwards() ;

```

---



---

**Algorithm 4:**  $safeDetection(p, in)$

---

```

1 foreach h ∈ next(p) do
2 | if acks(p, h) = ∅ then
3 | | foreach s ∈ match(p) where nh(s) = h do
4 | | | if pubs(h, s) ≠ ∅ then
5 | | | | append h to acks(p, in) ;
6 | | | | flagR(p, h) ;
7 | else
8 | | append h to acks(p, in) ;
9 | | flagR(p, h) ;

```

---



---

**Algorithm 5:**  $ackProcess(p, in)$

---

```

1 if acks(p, h) = {in}, ∀h ∈ match(p) then
2 | if h is a subscriber then deliver(h) ;
3 | else returnAck(p, h) ;
4 else
5 | remove in from acks(p, h) ;

```

---

$safeDetection(p, in)$  (Algorithm 4) ensures that there are no additional conflicts detected and that subscribers matching  $p$  are on separate links as other advertisements matching  $s$ .

Finally, we present the code executed upon receiving an acknowledgment when there are more acks required before delivering a publication to a given hop. If there are none, the buffered messages are delivered in the correct order (if the hop is a subscriber) or an acknowledgment is sent (if it’s a broker). Algorithm 5 shows how an acknowledgment for a publication  $p$  from  $in$  is processed.

In summary, the algorithm counts how many acknowledgments it is waiting for in order to deliver a publication along a certain hop. These acks ensure that the link is “flushed” with conflicting publications (possible due to the FIFO links) before delivering them in the correct order.

**Failures** — The algorithm is proven to be *safe* in a system with crash failures (see Theorem 5). Links are assumed to be reliable and FIFO (i.e., TCP). Messages can be lost at brokers and clients during failures. However, the set of publications actually delivered by each client has the same order, thus maintaining *weak pairwise total order*.

**Theorem 5.** *Given publications  $p$  and  $q$  from  $x_p$  and  $x_q$ , which are delivered to both  $s_1$  and  $s_2$  using paths  $P_1, P_2, Q_1$  and  $Q_2$ . Under the presence of failures, if  $s_1$  delivers  $p$  before  $q$ , then  $s_2$  only delivers  $p$  and  $q$  if and only if  $p$  is delivered before  $q$ .*

**Optimizations** — We offer two optimizations to our basic algorithm. One weakness of the original algorithm is that every ack must be received in order to deliver publications. A livelock situation can occur if the rate of new conflicting publications received is greater than the rate of acknowledgments received, since the queue will always be blocked as there are pending acks. Progressive delivery of the publications is possible by scanning from the head of the queue and delivering every subsequent publication until a non-acknowledged publication is reached. This subset of acknowledged messages must be sorted according to publication IDs, as usual.

Our second optimization seeks to minimize the detection overhead and sets a bound on the metadata attached to conflicting publications. For a given hop, the detection algorithm stops checking as soon as it determines that it is conflicting with another hop. The algorithm therefore determines every pair of hops  $\langle r, w \rangle$ , where the hop  $w$  is waiting for an acknowledgment to a request sent along  $r$ . During the *safeDetection* phase, the downstream brokers from the request side do not exactly know which advertisements and subscriptions are conflicting; instead they assume that advertisements for subscribers matching the publication are also overlapping with subscribers upstream. Likewise, brokers downstream from the waiting side do not know exactly which subscribers are conflicting: they simply assume every matching subscriber they have are conflicting and must be deferred.

There exists a tradeoff between detection overhead and false positive conflict rate. This has an impact on end-to-end delay of non-conflict subscribers and resolution overhead. On the other hand, publication size is fixed since no advertisement or subscription information is necessary; the request and wait flags are sufficient.

## V. EVALUATION

In this section, we experimentally evaluate our solution and its optimizations against two baselines: one which provides only per-publisher total ordering using FIFO links and another which uses a central sequencer. We also compare our approach in terms of performance to Spread [32], a group communication system.

**Setup** — The algorithms are implemented in Java as a module for the PADRES pub/sub prototype.<sup>3</sup> Our experiments run in two testbeds: a cluster of 24 machines each with four 1.86 GHz Xeon processors and 4 GB of RAM (referred to as PADRES-CLUSTER), and for larger experiments, on SciNet<sup>4</sup> using 96 GPC (General Purpose Cluster) machines. We use a synthetic workload, where each publisher sends a publication every 4 seconds and each publisher emits a single advertisement.

The network topology consists of several central brokers, connected in a chain. Each core broker is then connected to 5 edge brokers. This topology is modeled after interconnected data centers. Communication across data centers must then pass through the core brokers. For 96 brokers (on SciNet), there are 16 central brokers, each connected to 5 edge brokers. Advertisements and subscriptions are uniformly distributed across the edge brokers. In particular, every advertisement matches every subscription at a particular edge broker. This setup maximizes the number of delivery paths containing only one broker, which maximizes the number of potential conflicts. The publication rate, number of subscriptions, advertisements and interest overlap constitute the parameters which vary across the experiments.

**Metrics** — *Detection delay*: We measure the processing time required at each broker to detect conflicts for a publication. This is an important metric since the overhead is incurred on every subscriber, regardless of their conflict status. We seek to minimize this overhead as much as possible to limit the impact of our algorithm on non-conflicting subscribers.

*End-to-end delay*: This covers the detection and resolution overhead required before finally delivering the publication. The end-to-end delay with and without conflicts are compared to get an estimate of the resolution overhead. The metric only evaluates delivery of publications directly connected to the same broker as the subscriber: the impact of total order is greatest in those cases as the delivery without total order only involves one broker.

*Outgoing traffic*: We count the outgoing messages at each broker. This number is expected to increase with total ordering due to the presence of control messages.

*Ordering degree*: For the FIFO approach, we compute the ordering degree to evaluate how out-of-order the delivery stream is [33]. The publication ordering degree is computed as  $\sum_{D=0}^S (S-D)f(D)$ , where  $S$  is the length of the stream,  $D$  is the distance the publication was displaced,  $f(D)$  is the frequency (number of publications) which was displaced by distance  $D$ . This is then normalized by dividing by the maximum ordering degree possible,  $S^2$ . Total order provides a perfectly ordered stream that has a normalized degree of 1. The requirements for ordering is application-dependent.

<sup>3</sup><http://padres.msrg.toronto.edu/>

<sup>4</sup><http://www.scinethpc.ca/>

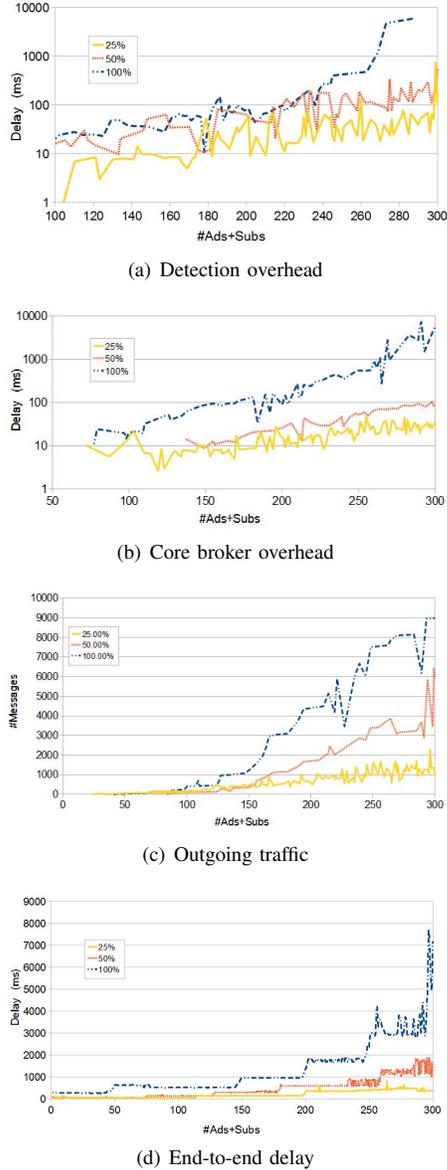


Figure 5. Progressive delivery algorithm

**Unoptimized algorithm** — The unoptimized algorithm waits for acknowledgments of every pending publication before delivering the whole buffer. Publication rate becomes an important concern here: a livelock can occur if the delivery queue fills up faster than the acknowledgment rate and no publications ever get delivered. In our setup, a rate of 120 conflicting publications/minute is enough to saturate the system, with three publishers and subscribers. Thus, no meaningful figures or data are produced for our regular solution as it simply does not scale with respect to publication rate.

**Progressive stream delivery algorithm** — This optimization avoids the livelock problem by delivering a subset of the pending queue at each acknowledgment. This is possible

by scanning the queue and delivering publications from the head of the queue until the next unacknowledged publication.

We measure the relative detection delay when varying the number of subscriptions and advertisements, as well as the fraction of overlapping publications on PADRES-CLUSTER. In other words, a 50% conflicting workload signifies that the publication will conflict with 50% of the subscriptions and advertisements. Furthermore, each edge broker contains only identical advertisements and matching subscriptions. This will reduce the number of brokers for each publication’s delivery tree. The publication rate also increases linearly in the number of subscriptions and advertisements.

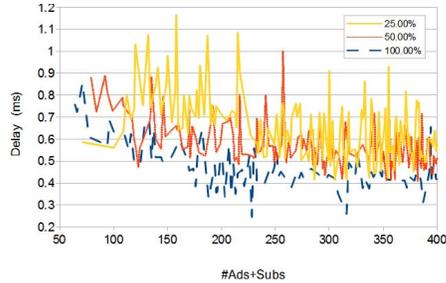
Fig. 5(a) shows that at 100%, the detection overhead can take up to 10 seconds at each broker when there are 300 combined subscriptions and advertisements at the broker. Fig. 5(b) shows the detection overhead at the core broker. The overhead for the 100% workload peaked at 7 seconds while for the other cases it is less than 100 ms. Since core brokers do not serve as edge brokers themselves, this suggests that processing and storing publications at the edge is more expensive. Fig. 5(c) shows the number of outgoing messages at a broker during a fixed period. The conflict percentage greatly affects the traffic volume through the number of acknowledgements generated, as demonstrated by the peak of 9000 messages for 100%, 6000 for 50% and 2000 for 25%. End-to-end delay also increases with the number of subscriptions and advertisements and is proportional to the fraction of conflicting publications, as shown in Fig. 5(d).

We also notice that with 100% conflict, some of our brokers will exceed the allotted Java heap space of 512MB when storing publications. This is due to the queue storing publications of unbounded size, since each publication must carry the entire set of advertisements and subscriptions it is conflicting with. Combined with the long end-to-end delay, the queues at the edge brokers are growing and consume more and more memory.

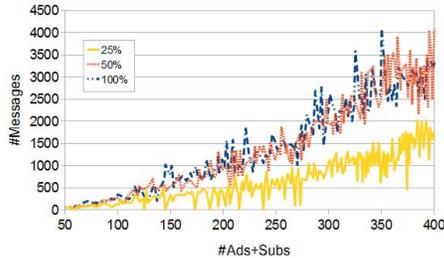
In summary, this optimization is not enough to achieve our desired scalability. The detection overhead is proportional to the number of advertisements and subscriptions since it must check every one of them for conflict at every broker. Furthermore, the publication message size is unbounded and depends on the number of conflicts.

**Fast detection algorithm** — The fast detection algorithm addresses the issues encountered with the previous algorithm. We focus on the performance of the fast detection algorithm by constructing a workload with no false positive (using the PADRES-CLUSTER). In Fig. 6(a), the overhead of the detection algorithm is proven to be stable over varying loads of subscriptions and advertisements, averaging less than 0.8 ms. Also note that the algorithm incurs more delay on average when there are fewer conflicts (e.g. 25% vs. 50%), since every advertisement and subscription must be checked on a given hop to conclude that there is no conflict.

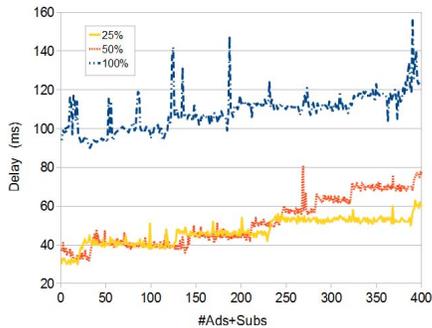
The traffic curve behaves similarly as in the previous



(a) Detection overhead



(b) Outgoing traffic



(c) End-to-end delay

Figure 6. Fast detection algorithm

algorithm, since it improves only detection time and the workload does not incur any false positives (see Fig. 6(b)).

The end-to-end delay also benefits greatly from the scalability of the fast detection algorithm. Fig. 6(c) shows that the end-to-end delay stays stable over an increasing number of advertisements and subscriptions. This is due to the uniform distribution of subscriptions and advertisements over the entire topology, which signifies that the number of brokers involved in the delivery tree of a publication does not change when more subscriptions are introduced.

#### Comparison with FIFO and Centralized Sequencer —

We compare the performance of our optimized algorithm using fast detection to two baselines. First, we look at the pub/sub system using only FIFO links, which maintains per-publisher ordering. Second, we implemented a central sequencing service. Publishers must request a sequence number before sending the publication. The setup for this experiment uses 800 subscriptions and 240 advertisements which are conflicting at a rate of 100% over a topology of

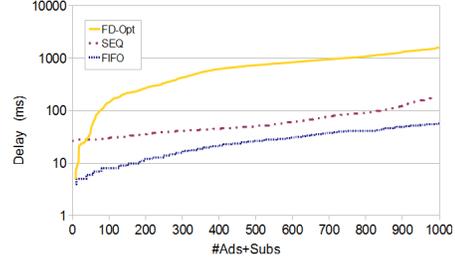


Figure 7. Baseline comparison: end-to-end delay

16 core brokers and 80 edge brokers running on SciNet.

In Fig. 7, the growth rate of end-to-end delay is linear to the number of subscriptions and advertisements in all cases: our solution (FD-Opt), per-publisher (FIFO) and central sequencer (SEQ). Although our overhead is the highest, it shows that our solution is scalable. However, the latency variance is much greater for our solution: publications have a maximum latency of 10 seconds, while remaining under 200 ms for the baseline solutions.

While the central sequencer performs better in terms of latency, the sequencer throughput becomes a bottleneck. *Throughput saturates at 130 publishers!* additional publishers are not able to send any publications because their sequencing requests are discarded. In comparison, all 240 publishers are functional in our solution’s experiments.

We evaluate the ordering degree of the FIFO solution by sampling random pairs of subscribers from different brokers and comparing their publication streams with each other. On average, conflicts were detected every 14 publications. The displacements were never more than 2. The normalized ordering degree was calculated to be 0.9952.

**Comparison with Spread** — Spread is a popular group communication toolkit written in C, notably used for distributed logging in Apache servers. It supports total order (referred to as *agreed ordering*) by circulating a token around the daemons [23]. We also compare against Spread without any ordering (called *reliable*).

As a group communication system, Spread fulfills a different role than pub/sub. Groups are at the same level of expression as topics in pub/sub. On the other hand, our content-based system would be analogous to having per-client filtering of messages within each group.

Brokers in our topology are mapped to daemons in Spread and can only communicate through unicast links (i.e., separate segments), as in PADRES. Furthermore, daemons in Spread are required to have complete knowledge of the domains, as they are arranged in a fully connected network. In contrast, our brokers can be arranged in any acyclic topology and are provided only with local information (e.g., direct neighbors). Note that Spread has a soft limit of 20 segments and therefore can only accommodate our edge brokers.

Experiments are conducted on PADRES-CLUSTER, with a 100% conflict rate. We measure the end-to-end latency as

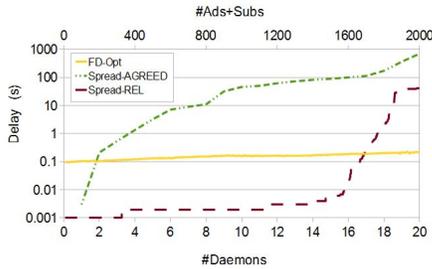


Figure 8. Spread comparison: end-to-end delay

the number of daemons (brokers) increases (see Fig. 8), with each daemon carrying 50 publishers and 50 subscribers.

With 1 daemon, Spread with total order (Spread-AGREED) performs better than our solution (FD-Opt), with less than 5 ms delay, since the token algorithm incurs no overhead. With 2 daemons, the delay immediately jumps to 200 ms, which is still comparable to the 100 ms provided by our solution. But we can quickly see that Spread does not scale to larger number of daemons, hovering at a latency of 700 seconds for 20 brokers and 1000 subscribers. In comparison, our solution remains in the sub-250 ms range.

To determine the impact of total ordering on Spread, we also measured its performance without any ordering guarantee. Although the performance of Spread-REL deteriorates rapidly after 16 daemons, the latency is still 41 s even at 2000 subscribers, which means the total order overhead increases the latency by a factor of 17. On the other hand, our solution produces an increase by a factor of 10.

## VI. CONCLUSIONS

This paper investigates the issue of total ordering in content-based pub/sub systems. The semantics of total ordering are defined and compared with topic-based pub/sub and other messaging models. The main contribution of this paper is a solution to the content-based pub/sub model using reliable FIFO channels. The solution does not require any global knowledge and is implemented directly in the brokers, eschewing the need for an external sequencing service. The solution leverages FIFO links as much as possible in order to contain the ordering overhead to publications and subscribers that truly require it. An evaluation of a real implementation of the algorithms shows that it scales well with the number of subscriptions and the publication size overhead is bounded.

## REFERENCES

- [1] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf, "Design and evaluation of a wide-area event notification service," *TOCS*, 2001.
- [2] G. Cugola, E. D. Nitto, and A. Fuggetta, "The JEDI event-based infrastructure and its application to the development of the OPSS WFMS," *TSE*, 2001.
- [3] I. Rose, R. Murty, P. Pietzuch, J. Ledlie, M. Roussopoulos, and M. Welsh, "COBRA: Content-based filtering and aggregation of blogs and RSS feeds," in *NSDI*, 2007.
- [4] K. Ostrowski and K. Birman, "Extensible Web services architecture for notification in large-scale systems," in *ICWS*, 2006.
- [5] G. Li and H.-A. Jacobsen, "Composite subscriptions in content-based publish/subscribe systems," in *Middleware*, 2005.
- [6] L. Fiege, M. Mezini, G. Mühl, and A. P. Buchmann, "Engineering event-based systems with scopes," in *ECOOP*, 2002.
- [7] C. Schuler, H. Schuldt, and H.-J. Schek, "Supporting reliable transactional business processes by publish/subscribe techniques," in *TES*, 2001.
- [8] T. Fawcett and F. Provost, "Activity monitoring: Noticing interesting changes in behavior," in *SIGKDD*, 1999.
- [9] Y. Tock, N. Naaman, A. Harpaz, and G. Gershinsky, "Hierarchical clustering of message flows in a multicast data dissemination system," in *IASTED PDCS*, 2005.
- [10] I. Koenig, "Event processing as a core capability of your content distribution fabric," in *Gartner Event Processing Summit*, 2007.
- [11] E. Fidler, H. A. Jacobsen, G. Li, and S. Mankovski, "The PADRES distributed publish/subscribe system," in *ICFI*, 2005.
- [12] P. R. Pietzuch and J. Bacon, "Hermes: A distributed event-based middleware architecture," in *ICDCS*, 2002.
- [13] J. Kienzle, C. Verbrugge, K. Bettina, A. Denault, and M. Hawker, "Mammoth: a massively multiplayer game research framework," in *FDG*, 2009.
- [14] A. R. Bharambe, S. Rao, and S. Seshan, "Mercury: A scalable publish-subscribe system for internet games," in *NetGames*, 2002.
- [15] G. Li, V. Muthusamy, and H.-A. Jacobsen, "A distributed service-oriented architecture for business process execution," *TWEB*, 2010.
- [16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*. Addison-Wesley, 1995.
- [17] G. Hohpe, "Programming without a call stack - event-driven architectures," <http://eaipatterns.com/docs/EDA.pdf>.
- [18] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *JACM*, 1996.
- [19] R. van Renesse, K. P. Birman, and S. Maffei, "Horus: a flexible group communication system," *CACM*, 1996.
- [20] T. H. Harrison, D. L. Levine, and D. C. Schmidt, "The design and performance of a real-time CORBA event service," *OOPSLA*, 1997.
- [21] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *CACM*, 1978.
- [22] X. Défago, A. Schiper, and P. Urbán, "Total order broadcast and multicast algorithms: Taxonomy and survey," *CSUR*, 2004.
- [23] R. Baldoni, S. Cimmino, and C. Marchetti, "Total order communications: A practical analysis," in *EDCC*, 2005.
- [24] C. Lumezanu, N. Spring, and B. Bhattacharjee, "Decentralized message ordering for publish/subscribe systems," in *Middleware*, 2006.
- [25] M. Kaashoek and A. Tanenbaum, "An evaluation of the Amoeba group communication system," in *ICDCS*, 1996.
- [26] H. Garcia-Molina and A. Spauster, "Ordered and reliable multicast communication," *TOCS*, 1991.
- [27] A. Schiper, K. Birman, and P. Stephenson, "Lightweight causal and atomic group multicast," *TOCS*, 1991.
- [28] L. L. Peterson, N. C. Buchholz, and R. D. Schlichting, "Preserving and using context information in interprocess communication," *TOCS*, 1989.
- [29] G. A. Wilkin, K. R. Jayaram, P. Eugster, and A. Khetrpal, "FaiDecs: Fair decentralized event correlation," in *Middleware*, 2011, pp. 228–248.
- [30] M. K. Aguilera and R. E. Strom, "Efficient atomic broadcast using deterministic merge," in *PODC*, 2000.
- [31] R. Guerraoui, "Genuine atomic multicast in asynchronous distributed systems," *TCS*, 2000.
- [32] Y. Amir and J. Stanton, "The Spread Wide Area Group Communication System," The Johns Hopkins University, Tech. Rep., 1998.
- [33] T. Banka, A. A. Bare, and A. P. Jayasumana, "Metrics for degree of reordering in packet sequences," in *LCN*, 2002.