Ahmad Baghdadi, Peter Amthor, Antonio Fernández Anta, and Boris Koldehofe. "ATOM: Asynchronous Total Order Mechanism for Publish/Subscribe", to appear in the Proceedings of the 28th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2026), October 9–11, 2026, Gothenburg, Sweden.

ATOM: Asynchronous Total Order Mechanism for Publish/Subscribe

Ahmad Baghdadi[1], Peter Amthor[2], Antonio Fernández Anta[3], and Boris Koldehofe[1]

[1] Philipps-Universität Marburg, Marburg, Germany
{ahmad.baghdadi,boris.koldehofe}@uni-marburg.de
[2] Technische Universität Ilmenau, Ilmenau, Germany
peter.amthor@tu-ilmenau.de
[3] IMDEA Software Institute & IMDEA Networks Institute, Madrid, Spain
antonio.fernandez@imdea.org

Abstract. Total order delivery semantics form the foundation for providing strong consistency guarantees in distributed applications. However, asynchronous many-to-many communication systems, such as current Publish/Subscribe systems, have been designed with different objectives in mind, supporting strong dynamics and high decoupling of application components. Therefore, available implementations of Publish/Subscribe only guarantee partial-order delivery of event messages. In this paper, we explore how to efficiently extend mechanisms deployed in popular Publish/Subscribe systems, such as Apache Kafka, to provide total order of events. Unlike previous attempts to build the order from scratch, this approach does not require any change to the underlying implementation. By proposing ATOM, a lightweight Asynchronous Total Order Mechanism, we show how to efficiently merge event partitions maintained in Publish/Subscribe to preserve time decoupling. We analyze the performance overhead of integrating ATOM into Kafka and compare it to a dedicated total-order messaging system called Multi-Ring Paxos.

Keywords: Total Order · Publish/Subscribe · Partitions.

1 Introduction

Distributed applications at scale, such as social media, e-commerce, and financial trading platforms, often must satisfy different consistency requirements across operational events within the same system. Weak consistency requirements, suitable for operations such as displaying social media feeds or browsing product catalogs, rely on *partial-order* semantics, in which causal relationships are maintained among event messages. In contrast, strong consistency requirements are essential for operations like processing payments or executing trade orders, demanding a *total order* where all nodes must agree on a single sequence of events that allows for deterministic computation and reproducible results.

Publish/Subscribe (*Pub/Sub*) is a widely deployed paradigm to support decoupled communication between event *producers* (*publishers*) and *consumers*

This version of the contribution has been accepted for publication, after peer review (when applicable) but is not the Version of Record and does not reflect post-acceptance improvements, or any corrections. The Version of Record is available online at: http://dx.doi.org/[insert DOI]. Use of this Accepted Version is subject to the publisher's Accepted Manuscript terms of use https://www.springernature.com/gp/open-research/policies/accepted-manuscript-terms.

Ahmad Baghdadi, Peter Amthor, Antonio Fernández Anta, and Boris Koldehofe. "ATOM: Asynchronous Total Order Mechanism for Publish/Subscribe", to appear in the Proceedings of the 28th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2026), October 9–11, 2026, Gothenburg, Sweden.

(*subscribers*) [12, 17, 21, 30]. For instance, the majority of top companies (80% of Fortune 100 companies) rely on Apache Kafka [5] in their IT infrastructure as one of the well-known Pub/Sub systems. However, Pub/Sub systems were not designed for strong consistency applications. Instead, they provide only weak consistency semantics in which subscribers deliver events of interest in a partial order. For example, Apache Kafka [5] orders events inside partitions of topics, Google Cloud Pub/Sub [19] enforces order using ordering keys, Amazon Kinesis [2] provides order per shard, and Amazon SNS [3] uses a special type of FIFO topics.

To meet strong consistency requirements, one solution is to build the total order of events from scratch and replace the deployed Pub/Sub system. This is, however, unacceptable for systems in production that rely on meeting flexibility and configurability requirements continuously. The other solution is to split the application architecture into different components that handle different consistency levels. That is, interfacing the deployed Pub/Sub system with additional dedicated total order mechanisms (e.g., consensus) to support strong consistency requirements. This interaction results in wasted system resources due to redundant operations that can be avoided (e.g., replication). In addition, it increases the complexity of integration and monitoring costs of the system.

This paper explores a lightweight, least-intrusive approach that extends existing deployments of Pub/Sub systems to support strong consistency requirements. We propose ATOM: Asynchronous Total Order Mechanism for Publish/Subscribe. ATOM builds on the partitioning model, which organizes events in FIFO order of the partitions, as in Kafka, and utilizes mechanisms that are currently supported (or abstracted) in most up-to-date systems mentioned earlier [2, 3, 5, 19]. ATOM adds complementary control information in the form of marker events (shortly *markers*) to all existing partitions. These markers allow subscribers to locally establish total-order delivery. ATOM does not apply any restriction on the publication of *normal events* that publishers send, since markers are injected into partitions asynchronously with respect to these events. Meanwhile, subscribers who desire to achieve total order use a deterministic merge mechanism to enforce cross-partition ordering based on marker events. Other normal subscribers have no overhead beyond skipping the marker events. Most importantly, all this is done without modifying the underlying Pub/Sub system.

The notion of total order considered here is a consistent delivery order of events among all subscribers that have common topics in their subscriptions. This order might not necessarily correspond to the actual publication order or adhere to a specific subscription of topics. The concept of total order is strongly related to group communication primitives such as *Atomic Broadcast* and *Total Order Broadcast* [13], and was initially introduced for fault-tolerant and consistent replication. Our contribution by ATOM is achieving total order in the context of Publish/Subscribe by: (1) isolating ordering events from the event publication and event replication phases, (2) lightweight integration with a minimum overhead to the underlying running system.

This version of the contribution has been accepted for publication, after peer review (when applicable) but is not the Version of Record and does not reflect post-acceptance improvements, or any corrections. The Version of Record is available online at: http://dx.doi.org/[insert DOI]. Use of this Accepted Version is subject to the publisher's Accepted Manuscript terms of use https://www.springernature.com/gp/open-research/policies/accepted-manuscript-terms.

Ahmad Baghdadi, Peter Amthor, Antonio Fernández Anta, and Boris Koldehofe. "ATOM: Asynchronous Total Order Mechanism for Publish/Subscribe", to appear in the Proceedings of the 28th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2026), October 9–11, 2026, Gothenburg, Sweden.

The remainder of the paper is organized as follows. section 2 presents the system model and assumptions. section 3 explains the concept of ATOM in detail. section 4 shows the evaluation results of ATOM. section 5 reviews related work. section 6 concludes the paper and discusses future directions.

2 System Model and Assumptions

A Pub/Sub system is an asynchronous communication model where publishers, subscribers, and brokers are process sets that communicate by sending and receiving events. These processes may fail (we assume a crash-recovery model) but do not behave arbitrarily (i.e., no Byzantine failures). The communication network is reliable. We assume a broker infrastructure that includes reliability mechanisms, such as event replication, acknowledgments, and retransmissions.

We assume a partitioning model where events published to topics are split into partitions distributed and hosted on brokers. A partition is a fundamental building block that accepts and stores published events for a durable period and allows subscribers to consume and replay them (see §2.1). We define the following sets: Publishers: $P = \{p_1, p_2, \ldots, p_n\}$, Subscribers: $S = \{s_1, s_2, \ldots, s_m\}$, Topics: $T = \{t_1, t_2, \ldots, t_r\}$, Partitions of topic $t_i \in T$: $PAR_i = \{par_1^i, par_2^i, \ldots, par_k^i\}$, Events of partition $par_j^i \in PAR_i$: $E_{ij} = \{e_1^{ij}, e_2^{ij}, \ldots, e_l^{ij}\}$, Subscribers of topic $t_i \in T$: $Sub(t_i) \subseteq S$.

We assume common interface operations such as: $advertise(t_i)$ to declare or create a topic t_i by an authorized publisher or a managing broker, $publish(t_i, E)$ to send a set of events E to topic t_i, $subscribe(t_i)/unsubscribe(t_i)$ to register and deregister interest in topic t_i, and $notify(E)$ to notify subscribers of new events.

2.1 Partition Properties

We assume the partition data structure is implemented as a *log* file written to the hosting broker's disk. It stores the actual event data, indexes, timestamps, and any metadata. Events are FIFO-ordered in the log (new events are only appended), and are identified by a sequential index number (i.e., the offset inside the log). Subscribers can only consume events if they are reliably stored in the log. The Pub/Sub system is assumed to satisfy the following properties:

Property 1 (Event Acceptance): An event e is accepted in partition par_j^i of a topic $t_i \in T$, if e is published by $p \in P$ and reliably written to par_j^i in a FIFO order, and has been assigned an offset $e.offset$ within that partition.
Property 2 (Unique ID): An event e accepted in the partition par_j^i of a topic $t_i \in T$ has a unique ID: $e.id = (t_i, par_j^i, e.offset)$, comprising the topic ID, partition number, and the offset in the partition.

We assume publishers are flexible in the partition used when publishing events to topics (e.g., fixed partition, round-robin). When events are received by the broker hosting the partition, they are accepted in arrival order (i.e., in the order

This version of the contribution has been accepted for publication, after peer review (when applicable) but is not the Version of Record and does not reflect post-acceptance improvements, or any corrections. The Version of Record is available online at: http://dx.doi.org/[insert DOI]. Use of this Accepted Version is subject to the publisher's Accepted Manuscript terms of use https://www.springernature.com/gp/open-research/policies/accepted-manuscript-terms.

Ahmad Baghdadi, Peter Amthor, Antonio Fernández Anta, and Boris Koldehofe. "ATOM: Asynchronous Total Order Mechanism for Publish/Subscribe", to appear in the Proceedings of the 28th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2026), October 9–11, 2026, Gothenburg, Sweden.

they enter the log's write buffer), which does not necessarily match the publication order. We assume the topic ID and partition number are defined by managing brokers when topics are created, or new partitions are added. Publishers and subscribers can inquire about this metadata by sending separate requests to brokers. Moreover, subscribers can request events by specifying the starting offset.

2.2 Total Order Properties

The total order properties of a Pub/Sub system can be mapped to those defined for total order broadcast [14], including two categories: ***liveness*** (expressed by validity) and ***safety*** (expressed by uniform integrity, agreement, and total order) as follows:

Validity: If $p \in P$ is a correct publisher, p published an event e to a topic $t_i \in T$, and e is accepted in $par_j^i \in PAR_i$, then every correct subscriber $s \in Sub(t_i)$ eventually delivers the event e.[4]
Uniform integrity (no creation or duplication): For every topic $t_i \in T$, each subscriber $s \in Sub(t_i)$ delivers an event e at most once, and only if e was previously accepted in the topic t_i by the system.
Uniform agreement: If any subscriber $s \in Sub(t_i)$ delivers an event e from the topic t_i, then every correct subscriber in $Sub(t_i)$ eventually delivers e.
Uniform total order: Let $s_i, s_j \in S$. If both subscribers deliver events e and e', then they deliver these events in the same order.

Following the classification in [10], we aim to match the *Strong Uniform Total Order*: without gaps in the sequence of delivered events among subscribers.

2.3 Reliability

Since total order properties relate directly to system reliability, we assume partition replication across brokers, and other mechanisms when producing and consuming events: acknowledgments, idempotent writes, and offset tracking.

For simplicity, partition replication can be done by primary-copy (other mechanisms are still compatible, such as consensus among brokers). One active partition (*leader*) serves publishers and subscribers, while replicas (*followers*) are synchronized with the leader. Leader election handles failures. When a leader fails, it becomes unavailable until a replica is elected as leader. Since a broker may host multiple leader partitions, all follow a similar scenario if the broker fails. Hence, the fault tolerance depends on the number of replicas per partition.

The leader broker sends acknowledgments to publishers after ensuring replication. Publishers maintain reliability by retransmitting events if acknowledgments are missing after a timeout. They use idempotent writing with a unique publisher ID and event sequence numbers to allow duplication detection. As acknowledgments, publishers receive the unique ID for each event after acceptance in any partition, which is used for tracking and recovery from failures.

[4] We assume that subscribers can replay accepted events by their unique IDs.

This version of the contribution has been accepted for publication, after peer review (when applicable) but is not the Version of Record and does not reflect post-acceptance improvements, or any corrections. The Version of Record is available online at: http://dx.doi.org/[insert DOI]. Use of this Accepted Version is subject to the publisher's Accepted Manuscript terms of use https://www.springernature.com/gp/open-research/policies/accepted-manuscript-terms.

Ahmad Baghdadi, Peter Amthor, Antonio Fernández Anta, and Boris Koldehofe. "ATOM: Asynchronous Total Order Mechanism for Publish/Subscribe", to appear in the Proceedings of the 28th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2026), October 9–11, 2026, Gothenburg, Sweden.

Subscribers obtain accepted events and their unique ID from the leader broker. They prevent event loss or duplication by tracking IDs after receiving, processing, and storing events. They use IDs to replay events after failure recovery.

The reliability separation between publishers, brokers, and subscribers allows graceful failure handling only after accepting published events in partitions hosted across brokers. Therefore, partition properties can guarantee validity, uniform integrity, and uniform agreement using previous mechanisms. However, an important system assumption is that, for topic $t_i \in T$, no sequencing order is defined among its partition set PAR_i, which means multiple subscribers of t_i could receive events from multiple partitions and deliver them in different orders. Also, no sequencing order exists among topics in the set T, which implies the same result when multiple subscribers receive events from multiple topics.

2.4 Out-of-order Delivery of Events

When subscribers consume events from multiple partitions (one topic with multiple partitions, or multiple topics with one partition), they may directly violate the total order property. Having different views on the event received order leads to different delivery decisions. For example, due to different event arrival times (e.g., network delays) and reading from partitions in different sequences, subscribers receive different event appearances and temporary gaps for newly accepted events. Since we assume by default that subscribers deliver events on a best-effort basis directly after receiving them, this immediately causes divergent delivery orders. In this work, we aim to introduce a total-order service that overcomes this issue and enables subscribers to deliver events in total order (i.e., $TOdeliver(E)$) to the application layer on top of this Pub/Sub system model.

3 Asynchronous Total Order Mechanism (ATOM)

This section introduces ATOM and its design properties. It applies to any Pub/Sub system that supports a partitioning model and the properties in section 2.

3.1 ATOM Overview

ATOM is designed to achieve total order in a lightweight form that preserves the scalability of the Pub/Sub system and uses the partitions architecture efficiently. Fig. 1 shows an overview of ATOM operations and indicates its two dedicated components. The first one is the *Marker Producer* (MP) on the broker side, and the second one is the *Total Order Enforcer* (TOE) on the subscriber side.

The MP is responsible for generating and maintaining marker events, which are periodically sent at rate λ_m to all partitions of all topics in the system. These markers enable all subscribers to make consistent decisions either to order and deliver events immediately or to temporarily queue them. As subscribers receive events from the partitions of the topics they are subscribed to, the TOE of each subscriber tracks and compares the received markers injected by the MP. The

This version of the contribution has been accepted for publication, after peer review (when applicable) but is not the Version of Record and does not reflect post-acceptance improvements, or any corrections. The Version of Record is available online at: http://dx.doi.org/[insert DOI]. Use of this Accepted Version is subject to the publisher's Accepted Manuscript terms of use https://www.springernature.com/gp/open-research/policies/accepted-manuscript-terms.

Ahmad Baghdadi, Peter Amthor, Antonio Fernández Anta, and Boris Koldehofe. "ATOM: Asynchronous Total Order Mechanism for Publish/Subscribe", to appear in the Proceedings of the 28th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2026), October 9–11, 2026, Gothenburg, Sweden.

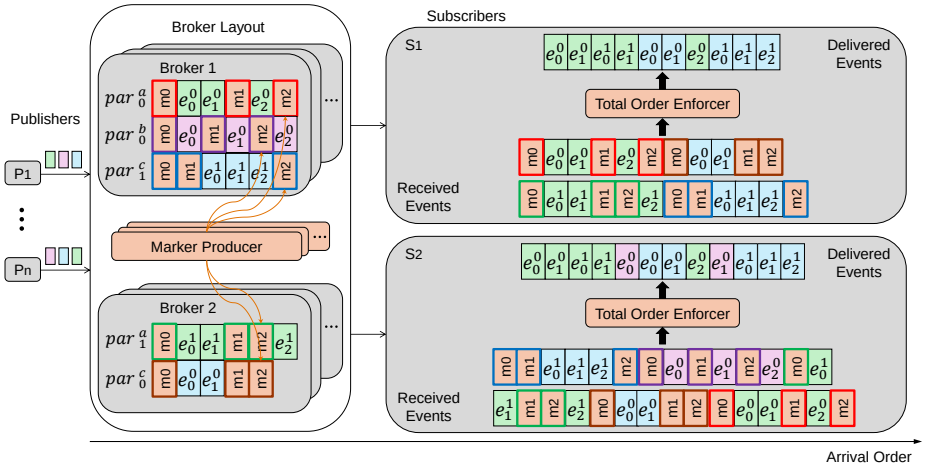


Fig. 1: ATOM Overview. The example shows a system with n publishers, two subscribers (S_1, S_2), and three topics: a (green), b (pink), and c (blue). S_1 subscribes to a, c and S_2 to a, b, and c. Topic a has par_0^a, par_1^a partitions, b has par_0^b, and c has par_0^c, par_1^c. The Marker Producer and brokers are replicated.

TOE determines the appropriate delivery sequence (via a deterministic merge mechanism), ensuring events are delivered in a unified, consistent order across all subscribers. In this way, ATOM ensures a global, consistent order of events over all partitions and topics. In the example of Fig. 1, S_1 and S_2 are subscribed to the common topics a and c, and both subscribers deliver the events of these topics in the same order, i.e., $e_0^0 \rightarrow e_1^0 \rightarrow e_0^1 \rightarrow e_1^1 \rightarrow e_0^0 \rightarrow e_1^0 \rightarrow e_2^0 \rightarrow e_0^1 \rightarrow e_1^1 \rightarrow e_2^1$.

3.2 ATOM Design Properties

Publishers In ATOM, the MP generates markers and injects them into partitions asynchronously with respect to the flow of normal events from publishers. This ensures that marker injection never blocks, delays, or interferes with event production operations. Therefore, publishers can continue sending events at maximum throughput without additional overhead (e.g., latency or backpressure).

Lemma 1 (Performance Isolation). *$\forall p \in P$, publication latency and throughput are independent of the marker injection rate λ_m and the marker processing cost, given (1) no shared blocking synchronization between the MP and publisher paths, (2) no mutable shared-data contention, and (3) no shared-resource competition. Hence, ATOM publisher performance equals the marker-free baseline.*

Proof. The publisher's path is disjoint from the MP path in: (1) control flow: the MP injects markers using an asynchronous non-blocking way, so publishers do not wait for any acknowledgment, callback, or state change before publishing normal events. (2) Shared data: markers are maintained locally by the MP and

This version of the contribution has been accepted for publication, after peer review (when applicable) but is not the Version of Record and does not reflect post-acceptance improvements, or any corrections. The Version of Record is available online at: http://dx.doi.org/[insert DOI]. Use of this Accepted Version is subject to the publisher's Accepted Manuscript terms of use https://www.springernature.com/gp/open-research/policies/accepted-manuscript-terms.

Ahmad Baghdadi, Peter Amthor, Antonio Fernández Anta, and Boris Koldehofe. "ATOM: Asynchronous Total Order Mechanism for Publish/Subscribe", to appear in the Proceedings of the 28th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2026), October 9–11, 2026, Gothenburg, Sweden.

separately from publishers, so they do not require locking operations. (3) Shared resources: the MP is a dedicated entity that has its own HW/SW resources that will not affect the service time of publishers. Therefore, the publication latency remains unaffected by λ_m, and the throughput saturates only at the broker limits. Hence, no added overhead is imposed on ATOM publishers.

The reliability of ATOM primarily depends on the MP (which is replicated, as shown in Fig. 1). A failure of this component does not disrupt event publishing across the entire system; it only delays event delivery for subscribers interested in the total order, while other subscribers can continue working normally.

Lemma 2 (Fault Isolation). *Under a failure state of the MP, the following properties hold:*

1. $\forall p \in P$, *the publishing operations continue to succeed without interruption.*
2. $\forall s \in S_{normal}$, *event delivery remains unaffected.*
3. $\forall s \in S_{ATOM}$, *event delivery is temporarily delayed but eventually resumed upon MP recovery. Previously published events are guaranteed to be delivered.*

Proof. The MP is decoupled from the normal-event replication path (i.e., partitions), so its failure does not block publishers from appending events. S_{normal} remain unaffected as they never interact with the MP. For S_{ATOM}, marker absence halts the progress (needed to establish total order). However, MP replication and persistent markers (as per the system architecture) ensure recoverability. Event delivery is delayed, not corrupted, and resumes upon MP restoration.

The proofs of Lemma 1 and 2 provide general principles that verify disjoint control, data, and resource contention, which ATOM satisfies by construction.

Subscribers Only subscribers that require a total order of events observe overhead by the TOE. The TOE performs event delivery in rounds, where the sequence number of a marker defines each discrete delivery round and the markers are the boundaries of these rounds. The TOE delivers events from topics and partitions in a predefined order (e.g., lexicographic). Other normal subscribers have no overhead beyond skipping the unneeded marker events. Similar conceptual abstractions can be found in distributed coordination, such as *synchronous rounds* or *synchrony view* [11, 16, 27], where events are sent, received, and processed in predefined time-based rounds. ATOM is flexible because it decouples the process of building the total order from the publish phase of normal events to the system. This reduces the overhead compared to any centralized ordering service, which would need to assign a unique sequence number to each event.

Definition 1 (Round-based Delivery). *Let $\Delta = m_i - m_{i-1}$ denote the duration of the interval between two consecutive markers (i.e., marker interval $= \frac{1}{\lambda_m}$). Let λ_E be the aggregate rate at which normal events are published and accepted in partitions. Consequently, the expected number of normal events buffered and processed by the TOE within a single round is $N(\Delta) = \lambda_E \cdot \Delta$. For any subscriber*

This version of the contribution has been accepted for publication, after peer review (when applicable) but is not the Version of Record and does not reflect post-acceptance improvements, or any corrections. The Version of Record is available online at: http://dx.doi.org/[insert DOI]. Use of this Accepted Version is subject to the publisher's Accepted Manuscript terms of use https://www.springernature.com/gp/open-research/policies/accepted-manuscript-terms.

Ahmad Baghdadi, Peter Amthor, Antonio Fernández Anta, and Boris Koldehofe. "ATOM: Asynchronous Total Order Mechanism for Publish/Subscribe", to appear in the Proceedings of the 28th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2026), October 9–11, 2026, Gothenburg, Sweden.

$s \in S_{ATOM}$, event delivery is performed in rounds. All events accumulated between marker m_{i-1} and m_i belong to round R_i (i.e., delivery period). Within a round, events are delivered in a predefined order (e.g., lexicographic).

The TOE utilizes a deterministic merge mechanism to enforce cross-partition ordering based on the appearance of marker events. Similar conceptual work is proposed in [1], which uses approximately synchronized clocks to predict the sending frequency of each publisher, then utilizes it for a consistent merging process in the network.

Lemma 3 (TOE Overhead). *For a subscriber $s \in S_{ATOM}$, Let K_s be the total number of partitions across all topics to which s is subscribed. For partition $j \in \{1, \ldots, K_s\}$, let $N_j(\Delta)$ be the number of events received from that partition during a given round R_i. The total number of events processed by the TOE in that round is: $N(\Delta) = \sum_{j=1}^{K_s} N_j(\Delta) = \lambda_E \cdot \Delta$. The worst-case computational cost incurred by the TOE for round R_i depends on whether events are already sorted by partition offset or not:*

1. ***Sorted case:*** $\mathcal{W}_{TOE}^{(sorted)}(R_i) = O\big(N(\Delta)\big)$.
2. ***Unsorted case:*** $\mathcal{W}_{TOE}^{(unsorted)}(R_i) = O\big(N(\Delta) \log N(\Delta)\big)$.

Proof. The TOE has three main operations: (1) Event buffering, (2) Marker detection, and (3) Event merging. Each event is appended to the buffer using an $O(1)$ operation. ***Sorted case:*** when events are already sorted by offset, the TOE checks each event once to detect the two markers. If the second marker is absent, events remain in the buffer until new ones arrive; the TOE resumes checking from where it previously stopped, so the marker-detection cost per event remains $O(1)$. If the second marker is present, the cost is still proportional to the number of newly checked events, which is a subset of the total events $N(\Delta)$. Once the markers are detected in all partitions, the TOE passes over the events between the markers to merge them into the delivery stream, incurring a cost of $O(N(\Delta))$. Removing the delivered events from the buffer has a similar cost to appending them. Therefore, the total execution cost for round R_i in this case is: $\mathcal{W}_{\text{TOE}}^{(\text{sorted})}(R_i) = O\big(N(\Delta)\big)$. ***Unsorted case:*** when events are not sorted by offset, the TOE must establish a total order before it can correctly detect markers and preserve per-partition semantics. It first sorts the buffered events of each partition by offset using a comparison-based sort. In the worst case, this costs: $\sum_{j=1}^{K_s} O\big(N_j(\Delta) \log N_j(\Delta)\big) \leq O\big(N(\Delta) \log N(\Delta)\big)$, where K_s is the number of partitions for subscriber s. After sorting, the TOE proceeds as in the sorted case, adding $O(N(\Delta))$ cost for merging and removal. The sorting cost dominates, yielding: $\mathcal{W}_{\text{TOE}}^{(\text{unsorted})}(R_i) = O\big(N(\Delta) \log N(\Delta)\big)$.

Lemma 4 (ATOM Overhead on Normal Subscriber). *For a subscriber $s \in S_{normal}$, the overhead of ATOM is strictly bounded by a constant-time filter. The event processing latency $\Lambda_{ATOM}(s)$ satisfies: $\Lambda_{ATOM}(s) = \Lambda_{baseline}(s) + O(1)$ where $\Lambda_{baseline}(s)$ is the latency in a marker-free system.*

This version of the contribution has been accepted for publication, after peer review (when applicable) but is not the Version of Record and does not reflect post-acceptance improvements, or any corrections. The Version of Record is available online at: http://dx.doi.org/[insert DOI]. Use of this Accepted Version is subject to the publisher's Accepted Manuscript terms of use https://www.springernature.com/gp/open-research/policies/accepted-manuscript-terms.

Ahmad Baghdadi, Peter Amthor, Antonio Fernández Anta, and Boris Koldehofe. "ATOM: Asynchronous Total Order Mechanism for Publish/Subscribe", to appear in the Proceedings of the 28th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2026), October 9–11, 2026, Gothenburg, Sweden.

Proof. Subscribers in S_{normal} do not require event ordering. When consuming from subscribed topics, they receive both normal and marker events. Since markers are clearly distinguishable via metadata, the subscriber inspects each event to discard markers. This filter is a single-branch operation applied without accessing the TOE. Hence, this overhead is independent of Δ, λ_{E}, and $N(\Delta)$.

Marker Producer To reduce the design complexity, the MP is a dedicated broker that acts as a standard event publisher. Its main task is to generate markers and publish them periodically to every topic partition in the system. Since the MP is oblivious to partition content when appending markers, its overhead is not influenced by the processing of normal events (i.e., by an increased number of partitions or event publishing rate). Moreover, in practice, marker messages are small compared to the typical size of normal events. Hence, the MP does not affect the overall system performance at scale.

Lemma 5 (MP Overhead and Performance at Scale). *The MP imposes a strictly bounded overhead on the system. As K is the number of partitions:*

1. *The complexity of periodic markers is $O(K)$.*
2. *The total storage and network overhead introduced by markers is bounded by K and independent of the normal event payload size.*
3. *As the system scales, the MP overhead does not affect overall performance.*

Proof. The MP issues exactly one append request per partition, each at $O(1)$ cost w.r.t. the number of normal events N. Therefore, the complexity per marker m is: $\mathcal{W}_{\mathrm{MP}}(m) = O(K)$, depending only on the marker size ms and decoupled from the average normal event size es. The overhead ratio is: $\frac{K \cdot ms}{N \cdot es}$. As N scales with publishing rate, or K with system scale, MP overhead stays $O(K)$ dominated. That is not a bottleneck and preserves performance at scale.

4 Evaluation

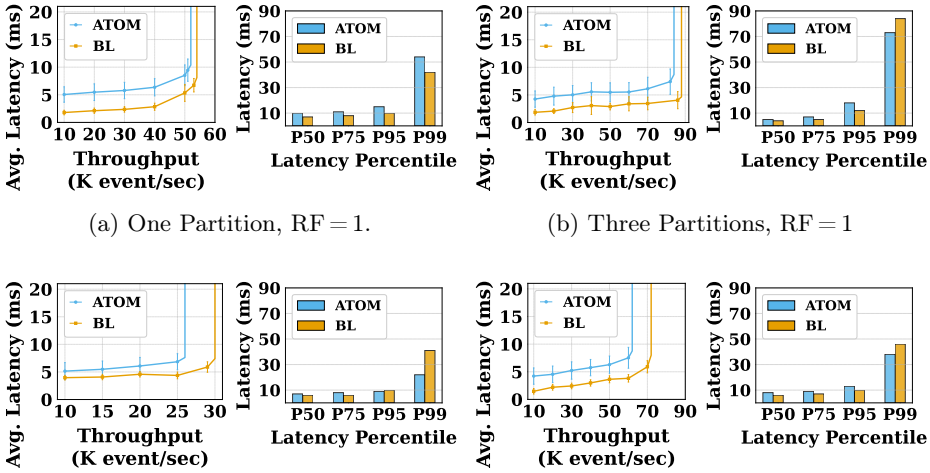
This section presents an experimental evaluation of ATOM, focusing on how the added markers and the TOE overhead affect throughput and end-to-end latency.

4.1 Setup

The experiments run on an 8-node cluster (Ubuntu 22.04, 5.4 GHz Core i7-13700K (16 CPU), 32 GB RAM, 1 TB Samsung SSD 980PRO, 1 Gbps Ethernet). The dedicated nodes are: one publisher, two MPs (active/standby), three brokers, two subscribers. ATOM is experimented on Kafka 3.9.1 (ZooKeeper mode) as the substrate with default configurations, except that the broker network and I/O threads were 4 and 12. We enabled all acknowledgments and idempotent writes for MP and disabled auto-commit for subscribers. To maintain repeatability and ensure system stability during experiments, the publisher event delay follows a

This version of the contribution has been accepted for publication, after peer review (when applicable) but is not the Version of Record and does not reflect post-acceptance improvements, or any corrections. The Version of Record is available online at: http://dx.doi.org/[insert DOI]. Use of this Accepted Version is subject to the publisher's Accepted Manuscript terms of use https://www.springernature.com/gp/open-research/policies/accepted-manuscript-terms.

Ahmad Baghdadi, Peter Amthor, Antonio Fernández Anta, and Boris Koldehofe. "ATOM: Asynchronous Total Order Mechanism for Publish/Subscribe", to appear in the Proceedings of the 28th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2026), October 9–11, 2026, Gothenburg, Sweden.



(a) One Partition, RF = 1.

(b) Three Partitions, RF = 1

(c) One Partition, RF = 2.

(d) Three Partitions, RF = 2.

Fig. 2: Latency vs. Throughput, and latency statistics at maximum throughput for ATOM and baseline (BL).

normal distribution around the mean delay of the desired production rate. Each experiment uses a 5 GB workload, 1 KB events split into 20 % warm-up and 80 % test events to reduce the initialization effects. Events are sent individually (i.e., not batched) to randomly selected partitions, resulting in a uniform distribution. Both subscribers receive and deliver events from all partitions.

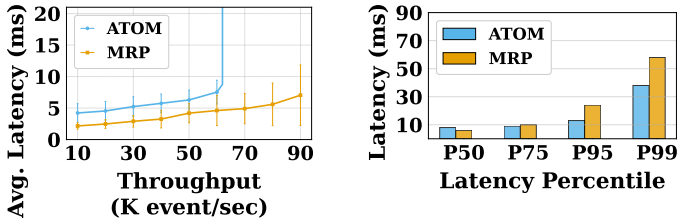
4.2 ATOM Performance

ATOM performance mainly depends on the marker interval Δ. However, other aspects, such as the publishing rate and the replication factor (RF), affect the performance of both ATOM and its substrate Kafka. Therefore, we compare the performance of Kafka, the *baseline (BL)* without event ordering, to ATOM, including both MP and TOE components. Since broker capacity is limited by the number of partitions, we tested two configurations: one partition on one broker, and three partitions across three brokers. The total order is achieved using one partition, but this shows the baseline limitation and the ATOM overhead. To show replication overhead, we tested RF = 1 (no replication) and RF = 2. We set marker interval $\Delta = 5$ ms (different values in §4.4), acknowledgment timeout $\Delta_{\text{ack}} = 500$ ms, and maximum retries $R_{\max} = 5$.

To show latency overhead, we tested different publishing rates and reported the average end-to-end latency for the test events after discarding the highest 5 % values. We considered stable system conditions in terms of event latency per partition and throughput: publisher throughput equals subscriber throughput. The maximum throughput resolution is 1 K event/sec. Fig. 2 shows latency evolution as throughput increases. We observe that the average latency behavior

This version of the contribution has been accepted for publication, after peer review (when applicable) but is not the Version of Record and does not reflect post-acceptance improvements, or any corrections. The Version of Record is available online at: http://dx.doi.org/[insert DOI]. Use of this Accepted Version is subject to the publisher's Accepted Manuscript terms of use https://www.springernature.com/gp/open-research/policies/accepted-manuscript-terms.

Ahmad Baghdadi, Peter Amthor, Antonio Fernández Anta, and Boris Koldehofe. "ATOM: Asynchronous Total Order Mechanism for Publish/Subscribe", to appear in the Proceedings of the 28th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2026), October 9–11, 2026, Gothenburg, Sweden.



ATOM: Three Partitions, RF = 2, $\Delta = 5$ ms. MRP: Three Rings, Quorum = 2.

Fig. 3: ATOM vs. MRP

in ATOM follows the baseline, with an increase below the selected value of Δ across all publishing rates. This is due to the nature of fetching events at subscribers, which allows multiple events to stack in broker partitions between fetch requests, and enables efficient event processing at the TOE. Optimized fetch configurations could improve the absolute results over the default.

The replication increases average latency. In ATOM, this increment is $\approx 85\%$ lower than the baseline, since it is included in the delay of the marker interval. Compared to one partition, this increment decreases when using three partitions, where the publication rate is split across partitions. However, the reduction is $\approx 12\%$ less in ATOM than the baseline, because of marker detection overhead.

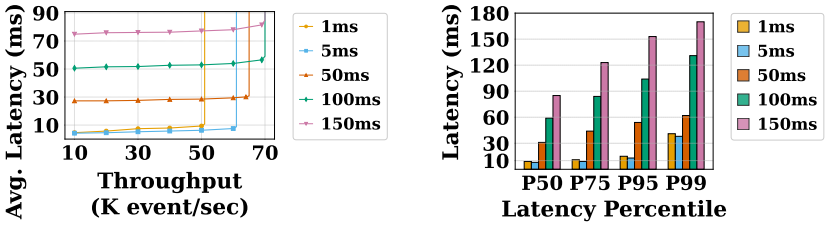
The replication decreases the maximum throughput. In ATOM, this decrement is $\approx 12\%$ higher than the baseline for one partition, and $\approx 44\%$ higher for three partitions. Compared to one partition, this decrement decreases when using three partitions. This reduction is $\approx 30\%$ less in ATOM than the baseline. Increasing the number of partitions will increase throughput until it saturates the system, resulting in unstable accumulated latency in some partitions. This is due to events queuing at the broker for one partition, and the saturation of the subscriber resources aggregating events from three partitions. Fig. 2 (d) shows that the maximum throughput of ATOM is $\approx 14\%$ lower than the baseline using $\Delta = 5$ ms. Also, the latency statistics show stability around 10 ms, except for the P99 tail latency, which indicates the start of a non-stable latency region.

4.3 ATOM vs. MRP

Multi-Ring Paxos (MRP) [23] is a protocol that can implement a standalone total order Pub/Sub system via multiple consensus instances. It has similar conceptual functionalities to Kafka, extended with ATOM. The roles in MRP can be mapped to ATOM as: *proposers* to publishers, *acceptors* to partition replicas, *coordinators* to leader partitions, *learners* to subscribers, and *rings* to partitions. In both approaches, the total order is achieved by utilizing a deterministic merge mechanism. Similar to markers in ATOM, *skip messages* in MRP are generated by the coordinator according to a predefined value of λ_{MRP}, which determines the maximum expected number of consensus in a time duration Δ_{MRP}. We configured MRP with three acceptors, one proposer, and two

This version of the contribution has been accepted for publication, after peer review (when applicable) but is not the Version of Record and does not reflect post-acceptance improvements, or any corrections. The Version of Record is available online at: http://dx.doi.org/[insert DOI]. Use of this Accepted Version is subject to the publisher's Accepted Manuscript terms of use https://www.springernature.com/gp/open-research/policies/accepted-manuscript-terms.

Ahmad Baghdadi, Peter Amthor, Antonio Fernández Anta, and Boris Koldehofe. "ATOM: Asynchronous Total Order Mechanism for Publish/Subscribe", to appear in the Proceedings of the 28th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2026), October 9–11, 2026, Gothenburg, Sweden.



Three Partitions, RF = 2.

Fig. 4: ATOM performance at different marker intervals.

learners that deliver events from three rings. The default configurations are used with in-memory mode, $\Delta_{\text{MRP}} = 1$ ms, $\lambda_{\text{MRP}} = 40$ K, and phase-1 resend time and proposer timeout = 60 sec. Using the same publishing rate control as in ATOM, the MRP proposer can send 90 K event/sec, then starts to have proposer timeout errors. Fig. 3 shows that MRP has comparable latency performance with ATOM using $\Delta = 5$ ms, but with a higher achievable system throughput. The latency statistics are comparable, except that MRP has higher tail latency than ATOM. The results confirm that MRP has optimized performance as a dedicated total order messaging system. However, using MRP requires either replacing the whole Pub/Sub system or interfacing with it at additional integration cost, while the main motivation for ATOM is to extend the system already in production.

4.4 Marker Interval

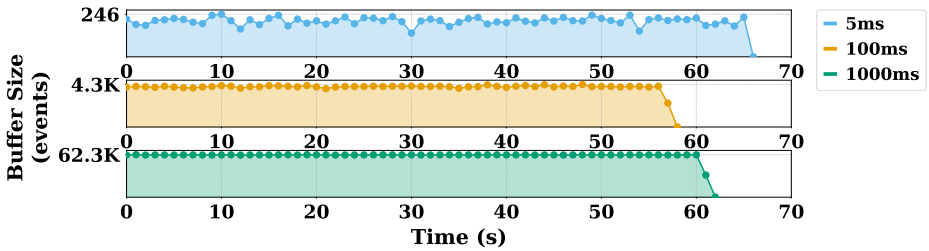
The marker interval Δ directly impacts the performance from three perspectives: (1) the added delay on event delivery, (2) the number of markers inserted in partitions (defines the delivery periods), and (3) the number of events between markers (defines the period size). Increasing Δ results in higher delivery delay, a smaller number of markers inserted between normal events (fewer delivery periods), and a larger number of events between markers (larger period size).

Fig. 4 shows ATOM's performance under different values of Δ between 1 ms and 150 ms (latency std. dev. omitted from the figure for clarity, the values are $\approx$25 - 30 % for both $\Delta = 1$ ms and $\Delta = 5$ ms, and $\approx$50 - 55 % for the remaining). As expected, the event delivery delay at TOE increases with higher values of Δ. However, when $\Delta = 1$ ms, the delay is higher compared to $\Delta = 5$ ms. That is, the overhead of detecting and processing a high number of markers is dominant. Therefore, we select $\Delta = 5$ ms as the minimum (optional) value.

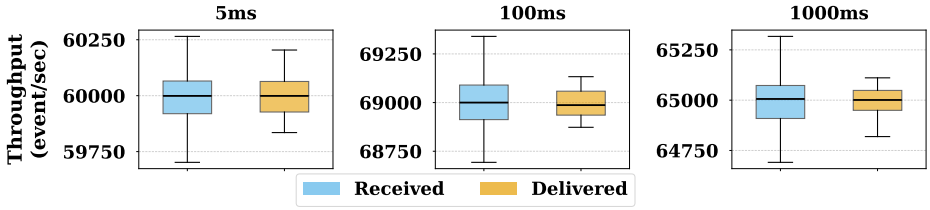
The number of markers in partitions is directly related to the Δ value (at fixed publishing rate). For the publishing rate of 10 K event/sec, the total markers from three partitions decrease from $\sim$300 K at $\Delta = 5$ ms to $\sim$10 K at $\Delta = 150$ ms. Also, the number of markers relates to the publishing rate (at a fixed marker interval). For $\Delta = 5$ ms, the total markers decreases from $\sim$300 K at the rate of 10 K event/sec to $\sim$50 K at the rate of 60 K event/sec. In both cases, the period size increases. Fig. 4 shows that the maximum throughput increases

This version of the contribution has been accepted for publication, after peer review (when applicable) but is not the Version of Record and does not reflect post-acceptance improvements, or any corrections. The Version of Record is available online at: http://dx.doi.org/[insert DOI]. Use of this Accepted Version is subject to the publisher's Accepted Manuscript terms of use https://www.springernature.com/gp/open-research/policies/accepted-manuscript-terms.

Ahmad Baghdadi, Peter Amthor, Antonio Fernández Anta, and Boris Koldehofe. "ATOM: Asynchronous Total Order Mechanism for Publish/Subscribe", to appear in the Proceedings of the 28th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2026), October 9–11, 2026, Gothenburg, Sweden.



(a) Buffer size in events at the end of each second.



(b) Received vs. Delivered throughput distribution.

Fig. 5: TOE buffer size and throughput distribution, $\Delta = 5$, 100, 1000 ms.

as Δ increases until the overhead of processing large periods dominates over processing fewer delivery periods. We notice that for $\Delta = 100$ ms and $\Delta = 150$ ms the maximum throughput is very close to the baseline. However, for $\Delta = 1000$ ms it drops to ~ 65 K event/sec. Therefore, selecting the best Δ value involves a trade-off between the selected delay and the maximum throughput achievable.

To evaluate the TOE buffering behavior, we select the case where the period size is maximized (i.e., at the maximum throughput) for three marker intervals: 5 ms (lowest delay), 100 ms (minimum delay with maximum throughput), and 1000 ms (highest added delay). Fig. 5 (a) shows the buffer size in events, starting from the first test event received and sampled at the end of each second until the experiment ends. This includes the non-delivered warm-up events and test events that are still being received and waiting to be delivered.

As expected, the buffer size increases with higher Δ values. As fetching events returns a varying number of events each time, not equal among partitions, and does not always represent a full period size. The buffer might not be fully cleared after each delivery period. For $\Delta = 5$ ms the average buffered value is ≈ 185 events/sec, increases to ≈ 4300 for $\Delta = 100$ ms, and ≈ 60500 for $\Delta = 1000$ ms. Fig. 5 (b) shows the statistical distribution of received and delivered events per second. The results are nearly identical for all Δ values, while the variation of delivered throughput decreases with higher buffering, which confirms the processing efficiency of the TOE.

This version of the contribution has been accepted for publication, after peer review (when applicable) but is not the Version of Record and does not reflect post-acceptance improvements, or any corrections. The Version of Record is available online at: http://dx.doi.org/[insert DOI]. Use of this Accepted Version is subject to the publisher's Accepted Manuscript terms of use https://www.springernature.com/gp/open-research/policies/accepted-manuscript-terms.

Ahmad Baghdadi, Peter Amthor, Antonio Fernández Anta, and Boris Koldehofe. "ATOM: Asynchronous Total Order Mechanism for Publish/Subscribe", to appear in the Proceedings of the 28th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2026), October 9–11, 2026, Gothenburg, Sweden.

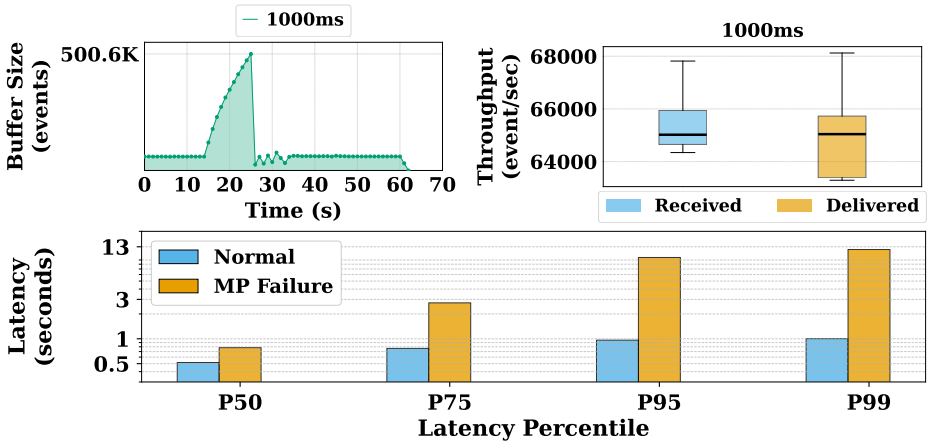


Fig. 6: MP Failure effect on TOE, $\Delta = 1000$ ms.

4.5 MP Failure Recovery

To investigate how MP failure affects ATOM performance, we crashed the active MP, allowing the standby to take over during the experiment. We modified two MP configurations: (1) heartbeat interval to 2000 ms, and (2) session timeout to 6000 ms, the minimum allowed in Kafka. These values manage failure detection and trigger reassignment of the replication topic to the other MP. We selected $\Delta = 1000$ ms with a 65 K event/sec publishing rate as a high buffering condition.

The first part of Fig. 6 shows a spike in buffer size when the active MP node crashes. Reassignment processes took ≈ 11 sec, including the first marker arrival from the new active MP. During this time, buffer size peaked at ≈ 500 K events. Upon receiving the first marker, the TOE processes all buffered events, and buffer size stabilizes around ≈ 60 K events after ≈ 8 sec. Hence, the throughput distribution has a high variation between received and delivered events, despite the similar median value. Obviously, the latency is affected, and it follows the delay of not receiving markers during the reassignment of the new MP.

The second part of Fig. 6 compares latency statistics between MP failure and the normal case without failure. Tail latency in both cases follows the maximum marker delay: 1003 ms in the normal case, and 12043 ms during MP failure. Although the reassignment process affects the delivery of $\approx 19\%$ of the events, we see effects on all latency statistics. For two reasons: (1) unstable buffering after the reassignment, which affects $\approx 13\%$ of the events, and (2) marker detection overhead at TOE, since the current implementation starts new detection from the latest marker's offset found. Further optimization is possible, especially when reaching a certain buffer level without finding markers. We conducted experiments modifying event size and distribution over partitions, but omitted results due to space limitations.

This version of the contribution has been accepted for publication, after peer review (when applicable) but is not the Version of Record and does not reflect post-acceptance improvements, or any corrections. The Version of Record is available online at: http://dx.doi.org/[insert DOI]. Use of this Accepted Version is subject to the publisher's Accepted Manuscript terms of use https://www.springernature.com/gp/open-research/policies/accepted-manuscript-terms.

Ahmad Baghdadi, Peter Amthor, Antonio Fernández Anta, and Boris Koldehofe. "ATOM: Asynchronous Total Order Mechanism for Publish/Subscribe", to appear in the Proceedings of the 28th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2026), October 9–11, 2026, Gothenburg, Sweden.

5 Related Work

The related work falls into two main categories: (1) group communication primitives, and (2) works leveraging traditional sequencing mechanisms in those primitives. The first category appears in toolkits such as JGroups [7], Appia [25], and Spread [31], or integrated into systems like CORBA event services [18, 26]. These implementations aim to simplify application development rather than optimize performance. The second category is well explained in [14], which uses dedicated sequencers (e.g., fixed or token-based), utilizes sender communication history, or relies on receiver-group agreement to establish the total order. This category has known limitations, as clearly noted in [24].

Existing solutions for total order in Pub/Sub systems rely mainly on the second category. However, direct deployment of these sequencing mechanisms is not efficient in this context. For instance, a centralized ordering service that provides a globally unique sequence ID for every published event. Redirecting all events to this service simplifies ordering but creates a single point of failure and a performance bottleneck as the number of events increases, limiting scalability. Moreover, it introduces additional latency from queuing delays or timeouts, directly impacting overall system throughput.

The ordering service can enable publishers to request and obtain a unique ID before publishing any event [8]. Alternatively, it can use a group of hierarchical sequencers [15, 22], a multistage ordering mechanism [9], or a synchronized shared logical clock between brokers [29]. All these approaches introduce communication, coordination, and group management overhead, which directly affects both scalability and throughput. In all, with many publisher requests, this becomes a performance bottleneck that does not scale well. Also, whenever a publisher sends an event, additional overhead is incurred in obtaining the unique sequence ID, increasing system latency and affecting scalability in terms of event number.

The service can aggregate events from multiple topics on-demand to reconstruct the total order. This can be done using stream processing tools such as Kafka Streams [6] or Apache Flink [4] on the subscriber side. However, these tools support only time-aware correlation (timestamps and time windows) not arrival time. Therefore, to achieve global total order across partitions, events still need an additional sequencer (e.g., a single partition) to preserve the order.

Consensus protocols [20, 28] can replicate the ordering service or be used among subscribers. For example, MRP [23] is an atomic multicast protocol that uses multiple independent Ring Paxos instances [24] to provide scalable total order. MRP can be utilized either by replacing the main components (i.e., publishers, brokers, and subscribers) of the deployed Pub/Sub system or by integrating it as an additional system, which incurs additional overhead.

6 Conclusion

ATOM is a lightweight mechanism for achieving total order delivery of events in Pub/Sub systems. It has two components: the Marker Producer (MP) and the

This version of the contribution has been accepted for publication, after peer review (when applicable) but is not the Version of Record and does not reflect post-acceptance improvements, or any corrections. The Version of Record is available online at: http://dx.doi.org/[insert DOI]. Use of this Accepted Version is subject to the publisher's Accepted Manuscript terms of use https://www.springernature.com/gp/open-research/policies/accepted-manuscript-terms.

Ahmad Baghdadi, Peter Amthor, Antonio Fernández Anta, and Boris Koldehofe. "ATOM: Asynchronous Total Order Mechanism for Publish/Subscribe", to appear in the Proceedings of the 28th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2026), October 9–11, 2026, Gothenburg, Sweden.

Total Order Enforcer (TOE), which can be simply integrated into the existing components of a Pub/Sub system without major modifications. Our evaluation shows that ATOM can achieve high performance, close to baseline (without event ordering), by tuning the marker interval to balance latency and throughput. In future work, we will investigate ATOM's performance at datacenter scale and explore ways to accelerate it by leveraging in-network computing capabilities. For example, we will examine how programmable switches can dynamically and efficiently determine the marker interval based on network state.

References

1. Aguilera, M.K., Strom, R.E.: Efficient atomic broadcast using deterministic merge. In: Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing. p. 209–218. PODC '00, Association for Computing Machinery, New York, NY, USA (2000). https://doi.org/10.1145/343477.343620
2. Amazon Web Services, Inc.: Amazon Kinesis, https://aws.amazon.com/Kinesis, accessed: 2026-07-25
3. Amazon Web Services, Inc.: Amazon Simple Notification Service (SNS), https://aws.amazon.com/sns, accessed: 2026-07-25
4. Apache Software Foundation: Apache Flink, https://flink.apache.org/, accessed: 2026-07-25
5. Apache Software Foundation: Apache Kafka, https://kafka.apache.org/, accessed: 2026-07-25
6. Apache Software Foundation: Kafka Streams, https://kafka.apache.org/documentation/streams/, accessed: 2026-07-25
7. B. Ban.: JGroups: A Toolkit for Reliable Multicast Communication, http://www.jgroups.org/, accessed: 2026-07-25
8. Balakrishnan, M., Malkhi, D., Prabhakaran, V., Wobbler, T., Wei, M., Davis, J.D.: CORFU: A shared log design for flash clusters. In: 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12). pp. 1–14. USENIX Association, San Jose, CA (Apr 2012)
9. Baldoni, R., Bonomi, S., Platania, M., Querzoni, L.: Efficient notification ordering for geo-distributed pub/sub systems. IEEE Transactions on Computers **64**(10), 2796–2808 (2015). https://doi.org/10.1109/TC.2014.2378287
10. Baldoni, R., Cimmino, S., Marchetti, C.: A classification of total order specifications and its application to fixed sequencer-based implementations. Journal of Parallel and Distributed Computing **66**(1), 108–127 (2006). https://doi.org/10.1016/j.jpdc.2005.06.021
11. Birman, K., Joseph, T.: Exploiting virtual synchrony in distributed systems. SIGOPS Oper. Syst. Rev. **21**(5), 123–138 (Nov 1987). https://doi.org/10.1145/37499.37515
12. Carzaniga, A., Rosenblum, D.S., Wolf, A.L.: Design and evaluation of a wide-area event notification service. ACM Trans. Comput. Syst. **19**(3), 332–383 (aug 2001). https://doi.org/10.1145/380749.380767
13. Chockler, G.V., Keidar, I., Vitenberg, R.: Group communication specifications: a comprehensive study. ACM Comput. Surv. **33**(4), 427–469 (Dec 2001). https://doi.org/10.1145/503112.503113

This version of the contribution has been accepted for publication, after peer review (when applicable) but is not the Version of Record and does not reflect post-acceptance improvements, or any corrections. The Version of Record is available online at: http://dx.doi.org/[insert DOI]. Use of this Accepted Version is subject to the publisher's Accepted Manuscript terms of use https://www.springernature.com/gp/open-research/policies/accepted-manuscript-terms.

Ahmad Baghdadi, Peter Amthor, Antonio Fernández Anta, and Boris Koldehofe. "ATOM: Asynchronous Total Order Mechanism for Publish/Subscribe", to appear in the Proceedings of the 28th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2026), October 9–11, 2026, Gothenburg, Sweden.

14. Défago, X., Schiper, A., Urbán, P.: Total order broadcast and multicast algorithms: Taxonomy and survey. ACM Comput. Surv. **36**(4), 372–421 (dec 2004). https://doi.org/10.1145/1041680.1041682
15. Ding, C., Chu, D., Zhao, E., Li, X., Alvisi, L., Van Renesse, R.: Scalog: seamless reconfiguration and total order in a scalable shared log. In: Proc. 17th Usenix Conference on Networked Systems Design and Implementation. p. 325–338. NSDI'20, USENIX Association, USA (2020)
16. Dwork, C., Lynch, N., Stockmeyer, L.: Consensus in the presence of partial synchrony. J. ACM **35**(2), 288–323 (Apr 1988). https://doi.org/10.1145/42282.42283
17. Eugster, P.T., Felber, P.A., Guerraoui, R., Kermarrec, A.M.: The many faces of publish/subscribe. ACM Comput. Surv. **35**(2), 114–131 (jun 2003). https://doi.org/10.1145/857076.857078
18. Felber, P., Garbinato, B., Guerraoui, R.: The design of a corba group communication service. In: Proc. 15th Symposium on Reliable Distributed Systems. pp. 150–159 (1996). https://doi.org/10.1109/RELDIS.1996.559713
19. Google: Google Cloud Pub/Sub, https://cloud.google.com/pubsub, accessed: 2026-07-25
20. Junqueira, F.P., Reed, B.C., Serafini, M.: Zab: High-performance broadcast for primary-backup systems. In: 2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks (DSN). pp. 245–256 (2011). https://doi.org/10.1109/DSN.2011.5958223
21. Koldehofe, B.: Fair adaptive overlay communications: A historical perspective on p2p-based publish/subscribe. In: From Multimedia Communications to the Future Internet. pp. 168–185. Springer (2024). https://doi.org/10.1007/978-3-031-71874-8_12
22. Lumezanu, C., Spring, N., Bhattacharjee, B.: Decentralized message ordering for publish/subscribe systems. In: Proc. ACM/IFIP/USENIX 2006 International Conference on Middleware. p. 162–179. Middleware '06, Springer-Verlag, Berlin, Heidelberg (2006). https://doi.org/10.1007/11925071_9
23. Marandi, P.J., Primi, M., Pedone, F.: Multi-ring paxos. In: IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012). pp. 1–12 (2012). https://doi.org/10.1109/DSN.2012.6263916
24. Marandi, P.J., Primi, M., Schiper, N., Pedone, F.: Ring paxos: A high-throughput atomic broadcast protocol. In: 2010 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). pp. 527–536 (2010). https://doi.org/10.1109/DSN.2010.5544272
25. Miranda, H., Pinto, A., Rodrigues, L.: Appia, a flexible protocol kernel supporting multiple coordinated channels. In: Proc. 21st International Conference on Distributed Computing Systems. pp. 707–710 (2001). https://doi.org/10.1109/ICDSC.2001.919005
26. Object Management Group: CORBA: Common Object Request Broker Architecture, https://www.omg.org/spec/CORBA/, accessed: 2026-07-25
27. Oki, B.M., Liskov, B.H.: Viewstamped replication: A new primary copy method to support highly-available distributed systems. In: Proc. Seventh Annual ACM Symposium on Principles of Distributed Computing. p. 8–17. ACM, New York, NY, USA (1988). https://doi.org/10.1145/62546.62549
28. Ongaro, D., Ousterhout, J.: In search of an understandable consensus algorithm. In: Proc. 2014 USENIX Conference on USENIX Annual Technical Conference. p. 305–320. USENIX ATC'14, USENIX Association, USA (2014)

This version of the contribution has been accepted for publication, after peer review (when applicable) but is not the Version of Record and does not reflect post-acceptance improvements, or any corrections. The Version of Record is available online at: http://dx.doi.org/[insert DOI]. Use of this Accepted Version is subject to the publisher's Accepted Manuscript terms of use https://www.springernature.com/gp/open-research/policies/accepted-manuscript-terms.

Ahmad Baghdadi, Peter Amthor, Antonio Fernández Anta, and Boris Koldehofe. "ATOM: Asynchronous Total Order Mechanism for Publish/Subscribe", to appear in the Proceedings of the 28th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2026), October 9–11, 2026, Gothenburg, Sweden.

29. Pereira, C.M.M., Lobato, D.C., Teixeira, C.A.C., Pimentel, M.G.: Achieving causal and total ordering in publish/subscribe middleware with dsm. In: Proc. 3rd Workshop on Middleware for Service Oriented Computing. p. 61–66. MW4SOC '08, ACM, New York, NY, USA (2008). https://doi.org/10.1145/1462802.1462813
30. Tariq, M.A., Koldehofe, B., Koch, G.G., Khan, I., Rothermel, K.: Meeting subscriber-defined QoS constraints in publish/subscribe systems. Concurrency and Computation: Practice and Experience **23**(11), 2140–2153 (2011). https://doi.org/10.1002/cpe.1751
31. Y. Amir, J. Stanton, O. Marth, and C. Tutu: The Spread Toolkit, http://www.spread.org/, accessed: 2026-07-25

This version of the contribution has been accepted for publication, after peer review (when applicable) but is not the Version of Record and does not reflect post-acceptance improvements, or any corrections. The Version of Record is available online at: http://dx.doi.org/[insert DOI]. Use of this Accepted Version is subject to the publisher's Accepted Manuscript terms of use https://www.springernature.com/gp/open-research/policies/accepted-manuscript-terms.