Decentralized Message Ordering for Publish/Subscribe Systems

Cristian Lumezanu, Neil Spring, and Bobby Bhattacharjee

University of Maryland
College Park, MD 20742
{lume,nspring,bobby}@cs.umd.edu

Abstract. We describe a method to order messages across groups in a publish/subscribe system without centralized control or large vector timestamps. We show that our scheme is *practical*—little state is required; that it is *scalable*—the maximum message load is limited by receivers; and that it *performs well*—the paths messages traverse to be ordered are not made much longer than necessary. Our insight is that only messages to groups that overlap in membership can be observed to arrive out of order: sequencing messages to these groups is sufficient to provide a consistent order, and when publishers subscribe to the groups to which they send, this message order is a causal order.

1 Introduction

Publish/subscribe (commonly, "pub/sub") is a useful design approach for large-scale distributed information dissemination applications. Pub/sub systems support loosely-coupled asynchronous communication between information producers and consumers. Producers (publishers) inject messages into the system, which routes messages to consumers (subscribers) that register interest in certain messages using subscriptions. In this paper, we present a protocol for providing an *ordered view* of messages sent in a pub/sub system. The order we provide is maintained across groups and users.

System Model Subscribers join *groups* that represent interests. The pub/sub system provides an API for nodes to **join** and **leave** groups, **send** messages to any group, and **receive** messages. Although it is reasonably easy to order messages to individual groups—simply elect a node to give each message a sequence number—ordering messages across groups is more challenging. Our ordering protocol enforces that the receive operation delivers messages in a consistent order across groups. More precisely, messages to groups that share subscribers are ordered so that the subscribers deliver messages to those shared groups in the same order.

1.1 Applications of Ordering

In the following applications, a centralized *coordinator* could order events. However, a single ordering authority limits feasible system size and introduces a single

point of failure or compromise. We assert that a distributed protocol, such as the one we present, enables large deployment.

Network games
Consider a multiplayer online game deployed using the publish/subscribe model [1], in which, for scalability, the virtual (game) world is divided into regions. Each player subscribes to the groups that represent nearby regions that it can affect or where events that can affect the player may occur [2, 3]. If multiple players have overlapping areas of interest, they must see the common events in the same order to maintain consistency.

Ordered message delivery provides game consistency. Consider three players that are near enough to each other that every event *published* by one player will be received by the other two players. If one player shoots and hits another, all should see the events in order, else physical rules are violated. Causal ordering is essential for game correctness. However, unrelated events in distinct regions need not be ordered.

Stock tickers
Consider an application in which messages correspond to stock market trades. Consumers at different brokerage firms may be interested in messages that satisfy different filters—by company size, geography, or industry, for example. The consumers will be members of groups based on their subscriptions, with every group receiving the same set of messages. An ordering protocol ensures that update operations that change state result in consistent states across the receivers that apply those updates in the same order.

Messaging
Internet messaging applications loosely follow the publish/subscribe model. For example, a user may choose to publish whether he is online or offline. Other users may subscribe to be notified of when a friend comes on-line by adding the user to their buddy list. A user may also join chat rooms (conferences) to converse with other users in the same rooms. Although ordering is not critical for "correctness" in messaging, enforcing that all messages appear in the same, likely causal, order should make such a system easier to use. For example, responses should always follow the messages to which they respond.

1.2 Overview of our Ordering Protocol

We distribute the task of ordering messages across *sequencing atoms*. Sequencing atoms assign sequence numbers to messages addressed to groups that share subscribers. Our approach is scalable because sequencing atoms order no more messages than the most active receiver in the network—sequencing atoms exist to order the intersections of group memberships, so do not order more messages than receivers. We separate the task of sequencing across as many sequencing atoms as possible for flexibility in distributing load, then rely on placing related

atoms on the same or nearby machines (*sequencing nodes*) to recover performance.

The insight that makes this possible is that the only destinations that can observe ambiguous order are those that subscribe to the same pairs of groups. Only messages to groups with at least two members in common must be ordered. By ordering those messages in the sequencing network and allowing unrelated messages to be ordered by end stations, we remove the requirement of centralized sequencing or long vector timestamps. The sequence numbers provided by sequencing atoms even allow events to be "committed" without ambiguity: receivers can tell when no prior messages are delayed.

For causal ordering, senders must subscribe to the groups to which they send. This requirement is simple and reasonable because receiving sent messages through the system also serves as an end-to-end reliability check. Our ordering is not total across all the users in the system: messages to unrelated groups may be delivered in any (perhaps globally inconsistent) order. Our distributed approach enables performance optimizations such as placing sequencers close to senders and receivers and trading message processing load against network load by combining sequencing atoms on the same node.

Our primary contribution is a method to order messages across groups of subscribers in a publish/subscribe system without centralized control. We present theoretical analysis to establish the correctness of our method and simulation results to verify its efficiency. Our broader goal is to develop primitives that improve the publish/subscribe model, that are scalable because they require no centralized servers or state, and that are practical by avoiding guarantees that applications do not need.

This paper is organized as follows. We survey related work in Section 2. We then describe the goals, assumptions, and procedures of our protocol in Section 3. We use simulations to measure performance in Section 4. We conclude in Section 5.

2 Related Work

The problem of ordered message delivery has been widely studied in distributed systems. Défago *et al.* [4] present an extensive survey, which we summarize here. Défago *et al.* organize algorithms by the assumptions they make on the underlying system (synchrony model, failure model, communication model, oracle model) and by the objectives they achieve. Here we focus on the ordering mechanisms.

Symmetric approaches are decentralized: each sender determines the order by appending information to all outgoing messages. The appended information reflects a causal order of messages, which may later be transformed into a total order using a predetermined function. Receivers use the attached information to decide whether to deliver or delay a message. Applications can append different types of information; most use timestamps or sequence numbers [5–9]. Including

this information in each message typically requires nodes to keep a view of the messages they have received and sent.

In asymmetric protocols, order is built by a sender, destination, or sequencer. In sender-based protocols [10–12], the sender can multicast a message only when granted the privilege, *i.e.*, when it holds a token. In sequencer-based approaches, typically one node is elected as a sequencer and is responsible for ordering messages [13–15]. More than one sequencer can be present, but only one will be active or relevant at a time [16, 17].

To preserve consistency among game states, networked multiplayer games enforce an unambiguous order of events. Typically, a centralized coordinator resolves all conflicts [18–21]. Although useful in a local area network, as the network grows, centralized approaches do not scale well and provide a central point of failure.

Although most work in decentralized ordering algorithms assumes only a single group, a few consider overlapping groups [14, 22–24]. Our approach is closest to that of Garcia-Molina *et al.* [14]. In the taxonomy of this section, their approach is asymmetric and sequencer-based: they order messages as they deliver them through a tree of subscriber nodes. A total order of messages results when messages traverse this tree, assuming, among other typical assumptions for fault-tolerant behavior, that message delay is bounded. The graph is arranged so that messages are sequenced by the destination nodes that subscribe to the most groups, and the task of sequencing messages is overlapped with distribution. We separate these tasks to sequencing atoms, which may be placed on any nodes in the network, and to a distribution tree, which may be tailored to perform well despite distant nodes. Our sequencing atoms sequence only messages for double-overlaps, in which groups share multiple members in common, not all messages for a destination. Although we provide only causal ordering, we expect that our design makes it possible for sequencing atoms to marshal fewer messages and do less work for each message.

There has been little interest in applying these ordering protocols in distributed publish/subscribe systems [25–28]. As the network grows, centralized approaches do not scale because the sequencer becomes a bottleneck and central point of failure. Furthermore, token-based protocols introduce long delays when nodes must wait for the token or recover lost tokens. Distributed approaches based on vector timestamps are more scalable but they incur prohibitive network overhead due to the large timestamps. Our protocol is both scalable and incurs low overhead. By distributing the task of sequencing across a network of sequencers, we remove the requirement for a centralized coordinator or large vector timestamps. Unlike vector timestamp approaches, the additional information we append to each message does not depend on the size of the destination group and is proportional, in the worst case, to the number of groups.

3 Ordering Protocol

Our model of an ordered message delivery system consists of three phases: *ingress*, where messages move from senders to the sequencing network, *sequencing*, where messages traverse sequencing atoms while collecting sequence numbers, and *distribution* where packets leave the sequencing network and are sent to destination nodes. We focus on sequencing; existing multicast delivery schemes can support ingress and distribution.

Our goal is to ensure a consistent ordered delivery of messages to members of the same groups. A group is formed of all subscribers that share a common subscription. Our key observation is that when messages are sent to groups with overlapping membership, receivers may make inconsistent decisions about the order of those messages. We call groups that have two or more subscribers in common *double overlapped*, and our approach is to provide a sequence number space for each double-overlapped set of groups. These sequence numbers remove the possibility of inconsistent ordering decisions by receivers. By sending messages through sequencing atoms arranged into a sequencing network, the network determines the order of related messages in a decentralized way.

The sequencing graph is arranged so that sequencing atoms (also called sequencers) instantiated for double-overlapped groups form paths that group messages can follow. A group may have many sequencing atoms because it may have many double-overlaps with other groups. The paths of messages addressed to doubly-overlapped groups intersect at the sequencer associated with the overlap, ensuring that these messages are ordered.

Sequencing atoms are virtual. They need not be placed on different hosts; in fact, placing atoms on the same host may improve performance. A *sequencing node* is a machine that hosts sequencing atoms. We assume that the group membership matrix—which nodes belong to which groups—is globally known; it can be kept in a distributed data store such as a DHT or it can be provided by the underlying publish/subscribe system.

3.1 Operation

Each sequencing atom maintains the following state:

- A sequence number for its overlapped groups.
- A group-local sequence number for the groups it acts as ingress node for.
- A forwarding table to direct messages to the next sequencer for each destination group.
- A reverse-path table listing the previous sequencer in the network for each group.
- An output retransmission buffer for each subsequent sequencer.
- A buffer to store received messages from previous sequencers.

Upon receiving a new message from outside the sequencing network, a sequencer assigns it a group-local sequence number. The message can be forwarded

immediately for distribution if its destination group has no double overlaps. Otherwise, if a group has a double overlap sequenced at this sequencer, the current sequence number for the overlap is added.

The message is then placed in the output buffer and transmitted to the next sequencer (if any) in the path for the group. The message can be removed from the buffer when this sequencer receives an acknowledgment from the next hop. We assume that there is a FIFO channel between any two sequencers. If the message is leaving the sequencer network, it will be sent to a delivery tree and on to group members.

This protocol provides two key properties. First, all members of the same group see messages in the same order, which is a causal order if the sender is also part of the group. Causal order expresses the "happens before" relationship among messages, as defined by Lamport [5]. Second, all destinations can make an immediate decision of whether to deliver or buffer arriving messages.

3.2 Sequencing Graph: Construction

The sequencing graph must meet two criteria:

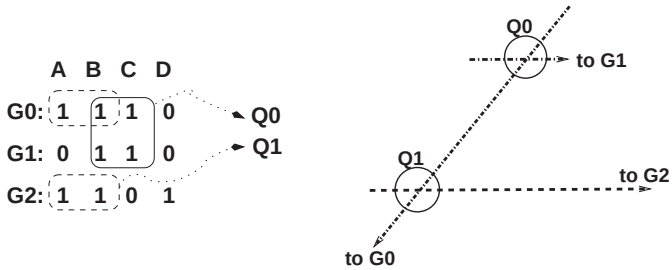
C1: *A single path must connect sequencers associated with each group.*
C2: *The undirected sequencing graph must be loop-free.*

C1 ensures that each message is sequenced relative to all other groups with which the destination group shares a double overlap. When leaving the sequencing network, each message has sufficient information that it can be ordered relative to the messages to all overlapping groups. **C2** prevents messages from having circular dependencies, *e.g.*, message a before b, b before c, and c before a. A loop in the sequencing graph could allow an atom to make an ordering decision inconsistent with the ordering of messages not seen by that sequencing atom, as we illustrate with an example in the next subsection. The group- and sequencer-based sequence numbers and ordered inter-sequencer message channels ensure a consistent order of related messages at destinations.

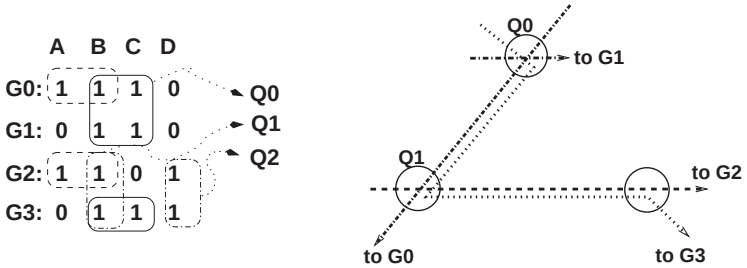
Operations on a sequencing network include adding, removing, and modifying groups. They correspond to the operations of adding, removing and changing a subscription in the publish/subscribe system. When a subscriber node A adds a new subscription, if there is no other node with the same subscription, a new group is created with A as its only member. Otherwise, A joins the group that is associated with the subscription. Similarly, when A removes one of its subscriptions, it will leave the group associated with the subscription. If A was the only member of the group, the group is deleted.

We describe only addition and removal of groups; changing the graph when group membership changes can be accomplished by adding a group with the new membership and removing the old one. Figure 1 illustrates these operations.

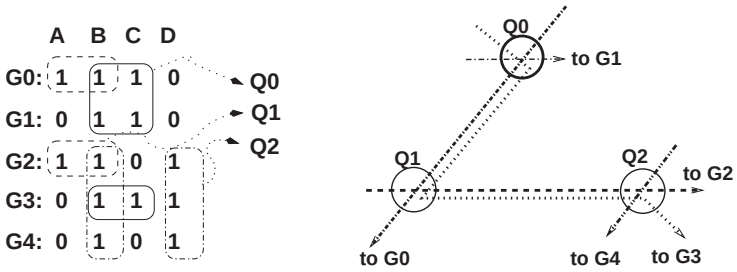
Adding the first group G0 is trivial: an ingress-only sequencer is created—this sequencer orders all messages sent to the group. When the second group, G1 is added, if the memberships of G0 and G1 overlap with at least two nodes (are



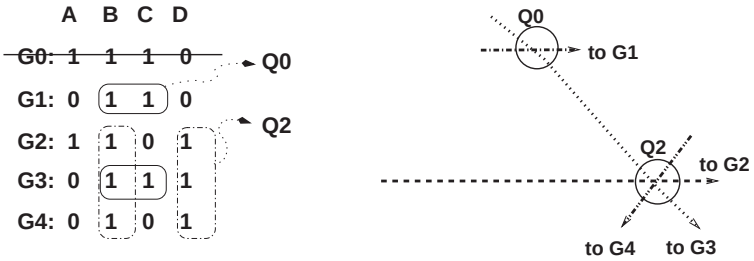
(a) Sequencing network for G0, G1 and G2



(b) G3 is added: Q0 and Q2 are associated to its overlaps; messsages to G3 are redirected through Q1 to avoid a loop in the sequencing graph



(c) G4 is added: messages to G4 traverse Q2 due to the overlap with G2 and G3



(d) G0 is removed: sequencer Q1 is no longer needed

Fig. 1. Adding and removing groups for a set of four nodes, A, B, C and D.

doubly-overlapped), a new sequencer, Q0, must represent G0 ∩ G1. All messages for both groups must transit this sequencer, and the G0-specific sequencer may be replaced or removed. This sequencer is *relevant* for all nodes in G0 ∩ G1; the rest need only use the group-local sequence number.

Adding each new group starts with the same basic procedure: a new sequencing atom is instantiated for any new double overlap. The new sequencing atoms must then be connected to the graph to form a path for the new group so that **C1** is satisfied. Unlike **C1**, **C2** is difficult to maintain using only local information. We use a global picture of the sequencing graph and subscription matrix state to find a new sequencer arrangement that satisfies **C1** and **C2**.

Removing a group may eliminate the overlaps that justify a sequencer's existence. Sequencers associated with a group can be removed lazily: adding ignored sequence numbers to a message does not hurt correctness, only efficiency. To remove a group, a termination message is sent to that group, signifying the end of the sequence space for that group, much like a TCP FIN. Each sequencer can inspect this termination message to determine if there is no longer overlap between the nodes this sequencer operates for. If the overlap is gone, the sequencer may retire by informing its parent to forward messages to its child for each sequenced group.

3.3 Sequencing Graph: Analysis

We present next an analysis of our protocol. We first describe how conditions **C1** and **C2** affect the sequencing graph and then we prove the unambiguity of the message delivery order across each group.

Let G be a group of subscribers that has double overlaps with other groups in the system. Each double overlap is associated with a sequencing atom and, according to **C1**, all the sequencers for the group form a single path in the sequencing graph. Since the graph is loop free and there are FIFO channels between each pair of sequencers, any order of arrival of two messages at a sequencing atom will be maintained by all the other sequencing atoms traversed by the messages afterward. We denote the sequence number assigned by sequencing atom Q to a message m addressed to group G by $Q(m)$ and the group-local sequence number with $G(m)$. The path of sequencing atoms traversed by a message m is $sp(m)$.

Definition 1. *Let G be a group with $|G| \geq 2$, let $A, B \in G$ and $\mathcal{M}_{A,B}$ be the set of messages received by both A and B. We define a relation $\leq_{A,B}$ on the set $\mathcal{M}_{A,B}$ such that $\forall m_1, m_2 \in \mathcal{M}_{A,B}$, $m_1 \leq_{A,B} m_2$ if and only if $Q(m_1) \leq Q(m_2)$ when $sp(m_1)$ and $sp(m_2)$ have a common sequencer Q, or $G(m_1) \leq G(m_2)$ otherwise.*

Theorem 1. *$\forall G, \forall A, B \in G, A \neq B$, $\leq_{A,B}$ is a total order.*

Proof. $\leq_{A,B}$ is a total order if it is *reflexive*, *transitive*, *antisymmetric* and *total*. For simplicity we refer to $\mathcal{M}_{A,B}$ and $\leq_{A,B}$ simply as $\mathcal{M}$ and $\leq_{\mathcal{M}}$.

Reflexivity: $\forall m \in \mathcal{M}$, $m \leq_{\mathcal{M}} m$.
The property is obviously true.

Transitivity: $\forall m_1, m_2, m_3 \in \mathcal{M}$ such that $m_1 \leq_{\mathcal{M}} m_2$ and $m_2 \leq_{\mathcal{M}} m_3$ then $m_1 \leq_{\mathcal{M}} m_3$.
Case I: If all three messages are addressed to the same group, and traverse a sequencing atom Q then $m_1 \leq_{\mathcal{M}} m_2 \Rightarrow Q(m_1) \leq Q(m_2)$ and $m_2 \leq_{\mathcal{M}} m_3 \Rightarrow Q(m_2) \leq Q(m_3)$. Therefore $Q(m_1) \leq Q(m_3)$ and consequently $m_1 \leq_{\mathcal{M}} m_3$. If the messages do not traverse a sequencing atom, transitivity is proved similarly using the group-local sequence numbers.

Case II: If two of the groups are identical and different from the third, there can be only one double overlap between them. All messages are sequenced by the sequencer associated with the overlap and the proof is identical to the first subcase of *Case I*.

Case III: We now consider the case when messages are addressed to three different groups and travel on sequencing paths $sp(m_1)$, $sp(m_2)$ and $sp(m_3)$. Since a group may have different double overlaps with each of the other two groups, the sequencing paths pairwise intersect. Therefore the paths of m_1 and m_2 must have a common sequencing atom, Q_1, which establishes the order between the messages. The same applies for m_2 and m_3 (both sequenced by Q_2) and for m_1 and m_3 (sequenced by Q_3). If the paths have more than one common sequencing atom, we pick the one closest to the sender as the most significant one. Because the sequencing graph is loop-free, it is imperative that $\mathsf{Q}_1 \subset sp(m_3)$, $\mathsf{Q}_2 \subset sp(m_1)$ or $\mathsf{Q}_3 \subset sp(m_2)$. We assume that $\mathsf{Q}_1 \subset sp(m_3)$—for the other two cases the reasoning is similar. Then, message m_3 transits Q_1 (although it does not receive a sequence number from it). From the hypothesis, we have $m_1 \leq_{\mathcal{M}} m_2$ and $m_2 \leq_{\mathcal{M}} m_3$, therefore $Q_1(m_1) \leq Q_1(m_2)$ and $Q_2(m_2) \leq Q_2(m_3)$. Because m_2 arrives before m_3 at Q_2 and because the order of arrival of two messages at all sequencing atoms on a path must be consistent, m_2 arrives before m_3 at Q_1. m_1 arrives before m_2 at Q_1 and, using the transitivity of the "arrives before" relation, it results that m_1 arrives before m_3 at Q_1. The consistent arrival order on $sp(m_3)$ maintains this property at Q_3, which will assign a lower sequence number to m_1. Since $Q_3(m_1) \leq Q_3(m_3)$ then $m_1 \leq_{\mathcal{M}} m_3$.

Antisymmetry: $\forall m_1, m_2 \in \mathcal{M}$, if $m_1 \leq_{\mathcal{M}} m_2$ and $m_2 \leq_{\mathcal{M}} m_1$ then $m_1 = m_2$.
If m_1 and m_2 travel through a sequencer Q then they will be assigned sequence numbers $Q(m_1)$ and $Q(m_2)$. If $m_1 \leq_{\mathcal{M}} m_2$ and $m_2 \leq_{\mathcal{M}} m_1$ then $Q(m_1) \leq Q(m_2)$ and $Q(m_2) \leq Q(m_1)$ and the total ordering of the natural numbers implies that $Q(m_1) = Q(m_2)$. A sequencing atom does not assign the same sequence number to two different messages therefore $m_1 = m_2$. If m_1 and m_2 do not traverse any sequencer they will be ordered based on the group local sequence number and the reasoning is the same.

Totality: $\forall m_1, m_2 \in \mathcal{M}$, either $m_1 \leq_{\mathcal{M}} m_2$ or $m_2 \leq_{\mathcal{M}} m_1$.
Any two messages received by both A and B can either be addressed to a single group or to two different groups. If their destination is a single group, the group local sequence number will be used to establish a total order between them. On the other hand, if they go to two different groups, they have to traverse the sequencing atom instantiated by the overlap (A,B). The assigned sequence numbers are used to determine the order. □

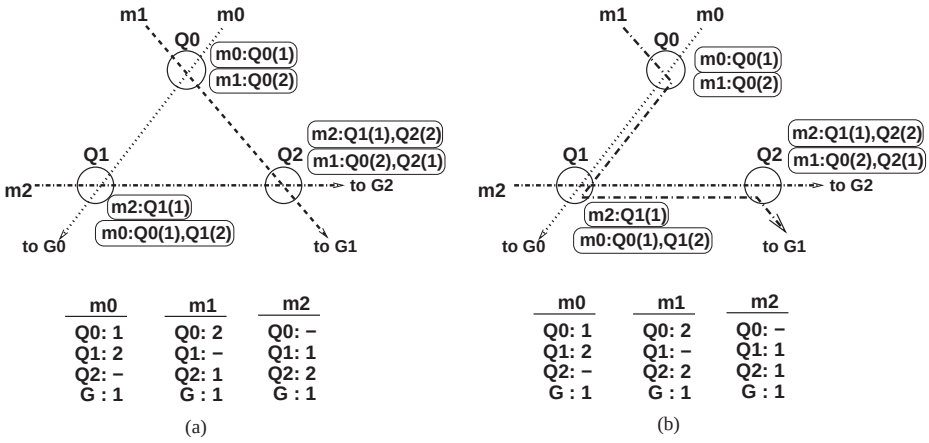


Fig. 2. Example of circular dependency among messages m_1, m_2 and m_3 and how it can be avoided; groups G0={A,B,D}, G1={A,B,C} and G2={B,C,D} are served by the sequencers Q0, Q1 and Q2. Each sequencer assigns a sequence number to each message that traverses it. (a) node B receives all three messages but cannot unambiguously decide on their order; (b) message m_1 is redirected through Q1 and the ambiguity is eliminated

Any destination node can make an instant and deterministic decision of whether to deliver an arriving message to the application or to buffer it. The order of delivery is consistent over all members of the same group, but it does not reflect causal relationships between messages. This is because the sender and the receivers are completely decoupled and the ordering is enforced by the sequencing graph. We achieve causal ordering only when the sender is part of the group to which the message is sent. This happens because only then the sender can be aware of any *a priori* causal relationship between messages and can propagate it across the sequencing graph.

As mentioned in Section 3.2, a sequencing graph must meet two criteria to unambiguously order messages. The first condition, **C1**, is easy to justify: multiple paths create nondeterminism which may produce ambiguous sequence numbers for messages. We illustrate the need for the second condition through the following example.

Consider four nodes A, B, C, D and three groups with the following memberships: G0={A,B,D}, G1={A,B,C}, G2={B,C,D}. Figure 2(a) shows the resulting sequencing network—without **C2**—with the sequencers labeled Q0, Q1, Q2. Now, assume that messages m_0, m_1, m_2 are sent to groups G0, G1, G2. Without **C2**, these messages will gain inconsistent sequence numbers, shown in the table in Figure 2(a), by the following process. Messages m_0 and m_1 both traverse sequencer Q0 and receive a sequencer number. If m_0 reaches Q0 first, it is tagged with sequence number 1 and m_1 tagged 2. Next, they continue on the path towards the destination group, m_0 sent to Q1 and m_1 to Q2. Meanwhile message m_2 is sent to G2 and reaches sequencer Q1 before message m_0. Thus, at Q1, m_2

is tagged with 1 and m_0 with 2. So far, message m_0 passed through both Q0 and Q1, being assigned sequence numbers 1 and 2. Because these were the only two sequencers through which it had to pass, m_0 can now be delivered to the members of G0, nodes A, B and D. Message m_2, on the other hand is forwarded to Q2. If the connection between sequencers Q1 and Q2 is very slow compared to the one between Q0 and Q2, m_2 reaches Q2 after m_1 does. Then, at Q2, m_1 receives sequence number 1 and m_2 sequence number 2. We show in the table from Figure 2(a) the sequence numbers of each of the three messages after they exit the sequencing network. Because each is the first and only message sent to its group, all have group sequence number 1. As the table shows, the three messages have a circular dependency. B cannot deliver m_0 because it waits for m_1, which cannot be delivered because of m_2, while m_2 depends upon the successful delivery of m_0.

We eliminate the circular dependency in Figure 2(b) by redirecting message m_1 through sequencer Q1 to make the sequencing graph loop free, condition **C2**.

3.4 Placing the Sequencing Atoms

Randomly scattering sequencing atoms throughout the network would lead to poor performance: because messages must traverse the path of sequencing atoms for the group, many needless network hops would result. We have developed a two-step heuristic for co-locating sequencing atoms on the same machines. The heuristic is based on the relationship between the double overlaps associated to the sequencing atoms. In the first step, we place on the same machine any sequencing atoms whose corresponding overlaps have a subset relationship between them. For example, let there be two sequencing atoms, Q1 and Q2, such that Q1 is associated to an overlap containing nodes A, B and C and Q2 corresponds to an overlap formed by A and B. Since $\{A,B\} \subset \{A,B,C\}$, Q1 and Q2 are co-located on the same node. In the second step of the heuristic, we also co-locate overlaps that do not have subset relationships between them but share at least a common node as follows. For each overlap, we choose at random one of its nodes, find all other overlaps that contain the chosen node and place the corresponding sequencing atoms on the same machine. We impose the restriction that each sequencing atom be co-located only once. This arrangement of sequencing atoms on the same sequencing node preserves our scalability goal—that no sequencing machine sees more messages than the most loaded receiver—without needlessly distributing related sequencing atoms throughout the network.

The selection of the machine on which to place a related set of sequencing atoms is also important. Ideally, we want to minimize the extra delay that a message experiences when it traverses the sequencing path. We abstract a related set of sequencing atoms by a sequencing node and we seek to find an optimal mapping between sequencing nodes and physical machines. We propose a simple heuristic that is run on behalf of each group as follows:

- if no sequencing node associated to the group has been assigned to a physical node yet, assign one at random

– if there are sequencing nodes already assigned to machines, then pick the closest unassigned sequencing node on their sequencing paths and assign it to neighboring machines.

The heuristic tries to put neighboring sequencing nodes on a sequencing path on close machines in the publish/subscribe infrastructure. This placement makes messages traverse relatively few extra hops to be ordered and helps us show that acceptable performance is feasible.

4 Results

In this section, we present simulation results to validate the performance of our ordering scheme. We only focus on the properties of the protocol when group membership is static or does not change very often.

4.1 Experimental Setup

We developed a packet-level discrete event simulator to evaluate the sequencing protocol. We simulated using a 10,000 node topology generated by GT-ITM [29]. The simulator models the propagation delay between routers, but not packet losses or queuing delays.

We attach hosts to the topology by grouping them into similar size clusters, then distributing each cluster uniformly at random through the topology. Nodes in the same cluster are placed close to each other. We choose this mapping because it is consistent with online communities, in which users tend to cluster around the lowest-latency server. We do not place any constraints on the publish/subscribe system that uses the ordering scheme. Messages travel from publishers to subscribers on the shortest path and any router in the topology can serve as a forwarding node. This is acceptable because our experiments are concerned only with the characteristics of the ordering layer. We are interested in measuring the penalty in performance that our primitive introduces with respect to the underlying layer. The mapping between the sequencing graph and the underlying infrastructure is done using the heuristic described in Section 3.4. Better heuristics may give better results—our intent in this section is to show that acceptable performance is possible.

We vary the number of end-hosts between 32 to 128, and each host can subscribe to zero or more groups. We vary the number of groups from 8 to 32. We rank the groups based on their size and we generate the size of each group using a Zipf distribution with exponent 1. The sizes are proportional to the function $r^{-1}/H_{n,1}$, where r is the rank of the group, n is the number of hosts and $H_{n,1}$ is the generalized harmonic number of order n of 1. We choose the Zipf distribution because it is known to characterize the popularity of online communities [30, 31]

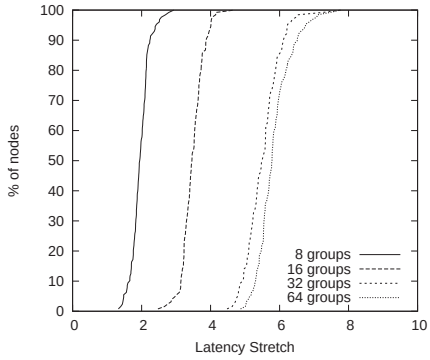


Fig. 3. Latency stretch for 128 subscriber nodes, when varying the number of groups.

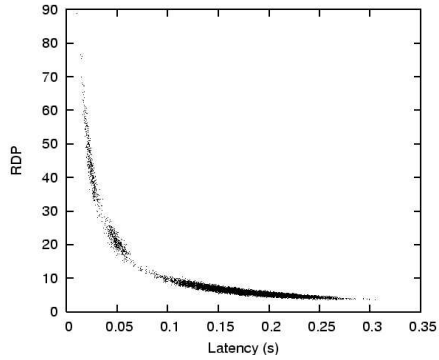


Fig. 4. Ratio between sequencing and unicast delay for each sender-destination pair versus the actual unicast delay, for 128 subscribers and 64 groups

4.2 Latency Stretch

We evaluate the extra delay messages encounter when traversing the sequencing network compared to taking the shortest unicast path. We measure the *latency stretch*: the ratio between the time taken for a message to traverse the network using the sequencers and the time taken using the direct unicast path. Similar metrics have been described by Chu *et al.* [32] (RDP) and Castro *et al.* [28] (RAD). RAD is defined per group and RDP per sender-destination pair; we believe latency stretch better represents the performance of our protocol because it captures the delay penalty of an individual node, when the node requires unambiguous delivery. To measure the latency stretch, each node sends a message to each of the groups it is part of, first using the sequencer network and then directly. We average the results and index them by destination nodes. We leave group membership fixed during the experiment.

Figure 3 presents the cumulative distribution of the latency stretch computed for 128 nodes subscribing to 8, 16, 32, and 64 groups. When there are fewer groups, the sequencing network is smaller and traversing it takes less time. For example, when we used 8 groups, latency stretch did not exceed 2.5. As the number of groups increases, so do the number of overlaps and the number of sequencing nodes that must be traversed. The growth is sub-linear: for 64 groups, the maximum latency stretch observed is less than 8. We quantify next how the increase in delay incurred by ordering is distributed with respect to the actual latency between the publisher and its subscribers. For this we compute the Relative Delay Penalty (RDP [32])—the ratio between the sequencing and unicast delay for each sender-destination pair—and plot it against the corresponding unicast delay between the sender and the destination. Figure 4 shows the results for 128 subscribers arranged in 64 groups. The highest values for

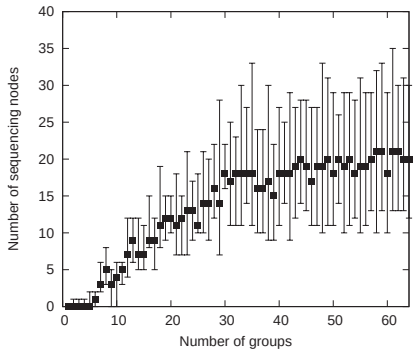


Fig. 5. Number of total sequencing nodes for 128 subscriber nodes, when varying the number of groups. Error bars indicate 10th and 90th percentiles.

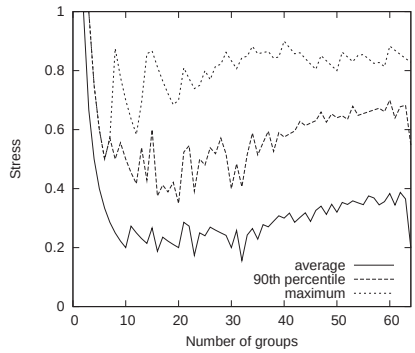


Fig. 6. Ratio between number of groups associated to a sequencing node and total number of groups, when varying the number of groups.

RDP correspond to the pairs in which the sender and the destination are very close to each other.

Increased delivery time in exchange for guaranteed order among messages is an inherent tradeoff of our approach. Delaying message delivery may be acceptable for Internet messaging or stock ticker applications, but generally it affects negatively the performance of network games. However, this section presents worst case results because we overestimate the performance of unicast: shortest unicast paths are rarely followed. To obtain faster delivery, the mapping of the sequencing graph should take into account the requirements of the applications that need ordering as well as the characteristics of the pub/sub infrastructure.

4.3 Sequencing Nodes

We next consider how adding groups affects the number of sequencing nodes and the stress on each node. We might worry that the number of sequencing nodes and the number of groups associated with each of them would increase exponentially as we add more groups; such a protocol would be impractical. To simplify presentation, we consider only the sequencing nodes that host non-ingress-only sequencers: each group has at most one ingress-only sequencer, so the ingress-only sequencers may grow at most linearly with the number of groups.

Figure 5 shows the average number of sequencing nodes created as we vary the number of groups. We vary the number of groups formed by 128 subscriber nodes from 1 to 64, and run the experiment 100 times. The error bars range from 10th to 90th percentile. As the number of groups increases, there are more overlaps and thus more sequencing nodes. After 30 groups, the number of sequencing nodes grows more gradually because many of the new overlaps have common members with existing overlaps, and so can be mapped to existing sequencing nodes.

We define the stress of a sequencing node as the ratio between the number of groups for which it has to forward messages and the total number of groups. A sequencer with a stress value of 1 forwards messages to all groups. In Figure 6, we present the average, 90th percentile and maximum values of stress as the number of groups increases. We observe the same behavior as in Figure 5. Initially, as we add more groups, we also add more sequencing nodes and the stress decreases and stabilizes around the value 0.2. After 30 groups, when the number of sequencing nodes increases more slowly, the stress slightly increases because there are more groups to be sequenced by the same number of sequencing nodes. The heuristic we used to map sequencing atoms to sequencing nodes makes sure that all the groups associated to a sequencing node share at least a member. As such, the load of this member is an upper bound for the load on any sequencing node that lies on the path to it.

4.4 Sequencing Atoms on a Path

Although the number of sequencing nodes remains small, the number of overlaps, and thus sequencing atoms, grows large. The size of the graph in atoms is less important, however, than the number of atoms each message must traverse, which represents how many sequence numbers a message must collect. Our approach is most attractive when the path length through the sequencing network is smaller than the number of nodes; that is, when the message overhead of sequence numbers provided by the sequencing network is less than that of system-wide vector timestamps. We compute the ratio between the number of sequencing atoms on a path and the total number of nodes, for different group sizes, and present it as a cumulative distribution in Figure 7. In the worst case, the number of sequencing atoms in the path of a message is less than half of the total number of nodes that participate. The path length through the sequencing network is bounded by the total number of groups, since a group can have an overlap with at most each of the other groups. As a result, our sequencer-based approach is attractive whenever the number of nodes exceeds the number of groups.

4.5 Varied Occupancy

Although we use a Zipf distribution to generate group sizes because we believe it models likely usage, we also wanted to explore worst-case usage scenarios. We define the expected occupancy as a measure of the density of the group membership. The value of the expected occupancy can be interpreted as the probability that a node is member of a group: an occupancy of 0 means that all groups are empty, while an occupancy of 1 means that every node subscribes to every group. Using 128 nodes and 32 groups, we vary the expected occupancy between 0 and 1 to see if the sequencing network approach is more efficient at some group densities.

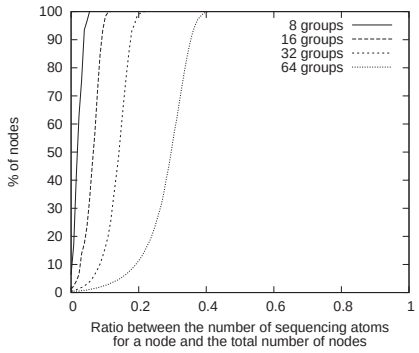


Fig. 7. Ratio between the number of sequencing atoms for each node and the total number of nodes, for 128 subscribers.

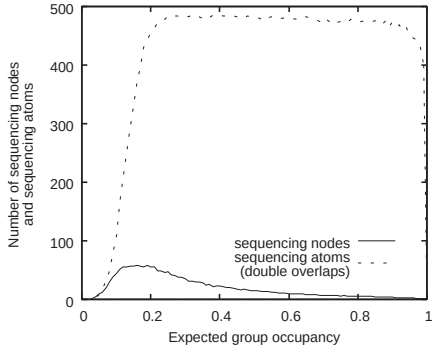


Fig. 8. Number of sequencing nodes and double overlaps vs. expected occupancy of groups, for 128 subscriber nodes and 32 groups.

Figure 8 illustrates how the expected occupancy of groups affects the average number of double overlaps and sequencing nodes. As the expected occupancy increases, the number of double overlaps and necessary sequencing nodes increase until approximately 0.2 occupancy. Beyond this, increasing group densities creates double overlaps that have common members with existing overlaps, and the number of sequencing nodes gradually decreases. When the group densities are very high (above 0.9), the overlaps include the entire population and the number of sequencing nodes drops to one.

5 Conclusion

Our primary contribution is a method for ordering messages in a pub/sub system without centralized control and without vector timestamps. We showed that it is practical and scalable, because little local and global state is maintained, because sequencing atoms can be placed to achieve good performance relative to a centralized sequencer, and because sequencing nodes order no more messages than destinations receive. Our insight is that only messages to groups with two or more common members must be ordered, and this provides a causal ordering when senders also subscribe.

This approach forms a new primitive for publish/subscribe systems. To investigate its applicability, we plan to apply the idea to the realistic workloads of these and other systems and measure when group membership is (or can be) geographically-correlated. We also intend to more completely understand the dynamic behavior of our algorithm. When changes in the group membership are infrequent or along existing patterns, we expect very little churn in the sequencing graph. However, we want to determine whether sequencing networks perform well even when incrementally updated as groups and nodes join and leave very often.

References

1. Bharambe, A.R., Rao, S., Seshan, S.: Mercury: A scalable publish-subscribe system for Internet games. In: NetGames. (2002)
2. Hu, S.Y., Liao, G.M.: Scalable peer-to-peer networked virtual environment. In chang Feng, W., ed.: NETGAMES, ACM (2004) 129–133
3. Morse, K.: Interest management in large-scale distributed simulations. Technical report, UC Irvine (1996)
4. Défago, X., Schiper, A., Urbán, P.: Total order broadcast and multicast algorithms: Taxonomy and survey. In: ACM Computing Surveys. (2004)
5. Lamport, L.: Time, clocks and the ordering of events in a distributed system. Communications of the ACM (1978)
6. Schiper, A., Eggli, J., Sandoz, A.: A new algorithm to implement causal ordering. In: 3rd International Workshop on Distributed Algorithms. (1989)
7. Peterson, L.L., Buchholz, N.C., Schlichting, R.D.: Preserving and using context information in interprocess communication. ACM TOCS **7**(3) (1989) 217–246
8. Dolev, D., Dwork, C., Stockmeyer, L.: Early delivery totally ordered multicast in asynchronous environments. In: 23rd Int'l Symposium on Fault-Tolerant Computing (FTCS-23). (1993)
9. Birman, K.P., Joseph, T.A.: Reliable communication in the presence of failures. ACM TOCS **5**(1) (1987) 47–76
10. Amir, Y., Moser, L.E., Melliar-Smith, P.M., Agarwal, D.A., Ciarfella, P.: The Totem single-ring ordering and membership protocol. ACM TOCS **13**(4) (1995) 311–342
11. Cristian, F.: Asynchronous atomic broadcast. In: IBM Technical Disclosure Bulletin. (1991)
12. Rajagopalan, B., McKinley, P.: A token-based protocol for reliable, ordered multicast communication. In: 8th Symposium on Reliable Distributed Systems (SRDS). (1989)
13. Kaashoek, M.F., Tanenbaum, A.S.: An evaluation of the Amoeba group communication system. In: ICDCS. (1996)
14. Garcia-Molina, H., Spauster, A.: Ordered and reliable multicast communication. ACM TOCS **9**(3) (1991) 242–271
15. Schiper, A., Birman, K., Stephenson, P.: Lightweight causal and atomic group multicast. ACM TOCS **9**(3) (1991) 272–314
16. Chang, J.M., Maxemchuk, N.F.: Reliable broadcast protocols. ACM TOCS **2**(3) (1984) 251–273
17. Whetten, B., Montgomery, T., Kaplan, S.M.: A high performance totally ordered multicast protocol. In: Selected Papers from the International Workshop on Theory and Practice in Distributed Systems. (1995)
18. Gautier, L., Diot, C.: Design and evaluation of mimaze, a multi-player game on the Internet. In: IEEE Int'l Conference on Multimedia Computing and Systems. (1998)
19. Ishibashi, Y., Tasaka, S., Tachibana, Y.: A media synchronization scheme with causality control in networked environments. In: IEEE LCN. (1999)
20. Ishibashi, Y., Tasaka, S., Tachibana, Y.: Adaptive causality and media synchronization control for networked multimedia applications. In: IEEE ICC. (2001)
21. Iimura, T., Hazeyama, H., Kadobayashi, Y.: Zoned federation of game servers: a peer-to-peer approach to scalable multi-player online games. In: NETGAMES. (2004)

22. Jia, X.: A total ordering multicast protocol using propagation trees. IEEE Trans. Parallel Distrib. Syst. **6**(6) (1995) 617–627
23. Ezhilchelvan, P.D., Macedo, R.A., Shrivastava, S.K.: Newtop: a fault-tolerant group communication protocol. In: ICDCS. (1995)
24. Aguilera, M.K., Strom, R.E.: Efficient atomic broadcast using deterministic merge. In: PODC. (2000)
25. Carzaniga, A., Rosenblum, D.S., Wolf, A.L.: Design and evaluation of a wide-area event notification service. ACM TOCS **19**(3) (2001) 332–383
26. Aguilera, M.K., Strom, R.E., Sturman, D.C., Astley, M., Chandra, T.D.: Matching events in a content-based subscription system. In: PODC. (1999) 53–61
27. Pietzuch, P., Bacon, J.: Hermes: A distributed event-based middleware architecture. In: 1st International Workshop on Distributed Event-Based Systems (DEBS'02). (2002)
28. Castro, M., Druschel, P., Kermarrec, A.M., Rowstron, A.I.: Scribe: A large-scale and decentralized application-level multicast infrastructure. IEEE Journal of Selected Areas in Communication (2002)
29. Zegura, E., Calvert, K., Bhattacharjee, S.: How to model an internetwork. In: IEEE Infocom. (1996)
30. Wolman, A., Voelker, G.M., Sharma, N., Cardwell, N., Karlin, A.R., Levy, H.M.: On the scale and performance of cooperative web proxy caching. In: SOSP. (1999)
31. Breslau, L., Cao, P., Fan, L., Phillips, G., Shenker, S.: Web caching and zipf-like distributions: Evidence and implications. In: INFOCOM. (1999)
32. Chu, Y.H., Rao, S.G., , Zhang, H.: A case for end system multicast. In: ACM Sigmetrics. (2000)