CAEVA: A Customizable and Adaptive Event Aggregation Framework for Collaborative Broker Overlays

Jianxia Chen, Lakshmish Ramaswamy
Department of Computer Science
The University of Georgia
Athens, GA 30602
Email: {chen, laks}@cs.uga.edu

David K. Lowenthal
Department of Computer Science
The University of Arizona
Tucson, AZ 85721
Email: dkl@cs.arizona.edu

Shivkumar Kalyanaraman
IBM Research India
Bangalore 560071 India
Email: shivkumar-k@in.ibm.com

***Abstract*—The publish-subscribe (pub-sub) paradigm is maturing and integrating into community-oriented collaborative applications. Because of this, pub-sub systems are faced with an event stream that may potentially contain large numbers of redundant and partial messages. Most pub-sub systems view partial and redundant messages as unique, which wastes resources not only at routers, but also at possibly resource constrained subscribers.**

In this paper, we present Caeva, a customizable and adaptive event aggregation framework. The design of Caeva exhibits three novel features. First, the tasks of merging messages and eliminating redundancies are shared among multiple, physically distributed brokers called aggregators. Second, we design a decentralized aggregator placement scheme that continuously adapts to decrease messaging overheads in the face of changing event publishing patterns. Third, we allow subscribers to choose a notification schedule that meets their specific needs. Results of extensive experiments show that Caeva is quite effective in providing flexibility and efficiency.

I. Introduction

The publish-subscribe (pub-sub) paradigm has been well-explored as an asynchronous, loosely-coupled communication mechanism for large-scale distributed systems [1], [2], [3], [4], [5], [6], [7], [8], [9]. Recently, there have been recognizable efforts towards adopting the pub-sub paradigm into community-oriented applications. Indeed, online social networks such as Digg [10] and Twitter [11] have incorporated pub-sub-style communication mechanisms.

However, the pub-sub substrates of most existing community-oriented applications are quite rudimentary. The design of pub-sub substrate *should* take into account the unique characteristics of collaborative communities. One such characteristic is the likelihood of inherently noisy event streams, including redundant publications, incomplete event messages, inaccurate event messages, and even events generated with malicious intent.

This paper describes the design, implementation, and performance of *Caeva*, which is a decentralized, dynamic, and configurable pub-sub system that handles redundant and partial events. *Caeva* uses a collaborative broker overlay to eliminate redundant messages (due to participants publishing event information that is already contained in one or more previously published messages) and merge same-event messages (due to multiple participants publishing messages with partial information about the same underlying event). By performing this task at the brokers, *Caeva* avoids placing this burden on the subscribers (who may be resource constrained in terms of power or bandwidth).

To operate effectively at a large scale, *Caeva* must address two key problems. First, aggregation must be decentralized, dynamic, and adaptive to achieve good performance, and the key to achieving this is developing an effective algorithm for placing aggregators within the broker overlay. Second, the ability of subscribers to control the inherent tradeoff between degree of aggregation and latency of notification is critical for usability.

Specifically, this paper makes three novel contributions.

- We present a collaborative event aggregation and redundancy elimination technique, in which event messages are aggregated in multiple stages and at multiple aggregators. Our technique includes decentralized protocols to coordinate the actions of various aggregators of an event so that subscribers receive notifications with low delay.
- We design and implement a distributed aggregator placement algorithm that continuously adapts to message publication patterns with the aim of

COLLABORATECOM 2010, October 9-12, Chicago, USA
Copyright © 2011 ICST
DOI 10.4108/icst.collaboratecom.2010.34

minimizing the message load within the overlay.

- We develop an efficient notification scheme for supporting subscriber-specified notification cycles.

We study the benefits and overheads of our scalable, decentralized mechanisms through series of experiments with particular attention to the broker overlay. The results demonstrate that the message load in *Caeva* system can be over 70% less than *Siena*, a similar system that does no message elimination.

The rest of this paper is organized as follows. Section II discusses the motivation behind our work. Then, we describe our aggregation scheme, its customization, and our flexible subscriber notification cycle scheme in Section III. The results are presented in Section IV. Finally, Section V describes related work, and Section VI summarizes the paper.

II. Motivation

Applications such as Twitter [11] incorporate a pub-sub-like substrate, in which the members of a (possibly ad-hoc) community or social group collaboratively report and receive events that may be of interest. In such applications with large user bases, several participants will likely notice an event simultaneously (or within a short duration of time) and report it to the system. Such a system is faced with an event stream with published messages that may have redundant data as well as partial event information. For example, in a collaborative traffic incident report system, different participants may report different aspects of an accident, with some reporting event information that has already been through one or more previous messages.

The simple strategy of relegating the partial and redundant message handling responsibilities to subscribers has several drawbacks, including: **(1)** useless event messages overwhelming low-end subscribers, which wastes bandwidth and power; and **(2)** significant communication overhead overwhelming the pub-sub system itself, which limits its scalability and performance. An alternative would be to perform these operations at centralized brokers (one broker per event) of the overlay [12]. However, the centralized brokers can quickly become overloaded. Further, relaying each published message to a centralized broker causes high messaging overheads within the overlay. Thus, in order to achieve scalability, the task of aggregating messages of an event should be shared by multiple brokers, and the set of brokers involved in aggregation should adapt to message publication patterns. In addition, the subscribers should be able to choose the degree of consolidation as per their needs and resource availabilities.

III. *Caeva*

Caeva is a collaborative, distributed-overlay based pub-sub infrastructure that supports event message aggregation and redundancy elimination in addition to routing messages from publishers to subscribers. Its design is motivated by *Agele* [12], which is described further in Section V. In this section, we first describe the architecture of *Caeva*. Then, we discuss decentralized, adaptive aggregation. Finally, we discuss customizing a notification schedule at the subscriber.

A. Architecture

Caeva is built upon a distributed overlay of message brokers (also referred to as *nodes*), represented as $\{b_1, b_2, \ldots, b_N\}$. Each broker is logically connected to a few other brokers such that the network forms a connected graph. The set of publishers and set of subscribers are represented as $\{p_1, p_2, \ldots, p_G\}$ and $\{s_1, s_2, \ldots, s_H\}$ respectively, with each publisher and subscriber connected to one of the brokers.

Caeva's subscription model is similar to type-and attribute-based pub-sub paradigm [7]. However, the proposed architecture as well as the associated techniques can be adapted to topic-based or content-based pub-sub systems. Every event in our system is associated with a *topic*, which provides a broad context for the event. For example, a traffic incident in a certain geographical area would represent a topic. In addition, events have a set of attributes (fields) that provide details of the event. The fields of an event e_q are represented as $\{e_q(1), e_q(2), \ldots, e_q(V)\}$. One of these fields (without loss of generality, the first field) is designated as the *event key*. The key field is descriptive, and it can be used in subscription predicates. For instance, the key for a traffic incident event would be the street intersection at which it occurs. Within a certain time-window, the key along with the topic corresponds uniquely to an event. The number of fields of an event, their types, and the key are determined by the event's topic. Subscriptions are specified with respect to the event topic as well as its fields. A subscription has to necessarily identify the topic of interest. Additionally, it *may* specify predicates involving the fields associated with the topic.

There can be multiple published messages associated with a single event (represented as $\{e_q^1, e_q^2, \ldots, e_q^U\}$ for event e_q) , possibly published by multiple publishers. Each message contains a subset of fields of the corresponding event. The fields of an event message e_q^r are represented as $\{e_q^r(1), e_q^r(2), \ldots, e_q^r(V)\}$. According to key-topic uniqueness assumption, if the first message of an event is published at time t_f, any messages with an identical key-topic pair generated between t_f and $t_f + W$ correspond to the same event. Publishers may *advertise* the types of events they are going to generate. However, the system can be configured to work without advertisements, in which case it is assumed that every publisher can publish all types of events.

Similar to many existing pub-sub systems [2], [3], *routing acyclic graphs (AGs)* comprised of brokers from the overlay form the basis for routing events from publishers to subscribers. Routing AGs are constructed in a completely decentralized fashion by peer-to-peer forwarding of subscriptions and advertisements. The predicates of subscriptions with the *same* topic are aggregated at brokers using the subsumption relationship, and a more generic subscription is forwarded. While *Caeva* maintains a distinct routing AG for each topic, individual brokers can belong to multiple routing AGs.

B. Decentralized, Adaptive Aggregation

Caeva uses a collaborative, decentralized and adaptive approach to aggregating events and eliminating redundancy. At a high-level, decentralized aggregation has a resemblance to the operator placement problem in distributed stream processing systems [13]. The question, therefore, is whether similar techniques can be used for the problem at hand. However, as we discuss in Section V, in a community-based event system, message publishers (source nodes) of a particular event are not known before hand, which precludes adopting heavyweight, plan-based techniques that have been used for distributed stream processing systems. We need a lightweight and dynamic scheme that does not need apriori knowledge of message sources of an individual event.

In our approach, designated brokers within the routing AG of a particular event type participate in aggregating and eliminating events of that type. Such brokers are referred to as *aggregators*. Each aggregator is autonomous and maintains a buffer that stores part of an event.

In *Caeva*, we coordinate the activities of the various aggregators of an event. This ensures that subscribers receive event information available in one composite message at the end of each notification cycle. A subset of aggregators, called *active aggregators* (AAs), additionally perform coordination. One of the active aggregators, the *coordinator*, coordinates the final round of aggregation and routes the aggregated message to subscribers. We denote all non-active aggregators as *passive aggregators* (PAs). *The key to* Caeva *is that the aggregators are chosen dynamically, and then are moved adaptively when necessary.*

In the next two subsections we explain the operations of active and passive aggregators and the coordinator. In turn, we discuss the dynamic aggregation within *Caeva*, its coordination algorithm for the active aggregator, and then how aggregators are placed within the broker overlay and moved adaptively.

In this discussion, we focus on the routing AG of a single event type. However, multiple routing AGs can simultaneously exist in *Caeva*, and the techniques and mechanisms discussed below apply to the routing AGs within the broker overlay. For now, we assume all subscribers have the same notification cycle duration; the next section relaxes this assumption.

Notationally, the set of passive aggregators is denoted $PvSet = \{pv_1, pv_2, \ldots, pv_F\}$ and its active aggregator set $AvSet = \{av_1, av_2, \ldots, av_G\}$. The coordinator of the event e_q is represented as C_q.

Dynamic Aggregation: When the event message e_q^r reaches a passive aggregator pv_f, there are three possible cases: **(1)** pv_f has a message corresponding to the event e_q in its buffer, and that message is a superset of all the fields contained in e_q^r. In this scenario, e_q^r is redundant and therefore dropped. **(2)** pv_f has a message pertaining to event e_q in its buffer, but that message does not have all the fields contained in e_q^r. In this case ,e_q^r is merged with the buffered message. **(3)** e_q^r is the first message of event e_q. Here, pv_f inserts it into its buffer, but also passes it to its upstream neighbor; it will eventually reach an active aggregator. PA pv_f will eventually get a reply back from the active aggregator indicating the coordinator and notification cycle. pv_f sends the (partially) aggregated message in its buffer to C_q just before the end of every notification cycle (the manner in which pv_f discovers C_q and the duration of e_q's notification cycle is discussed later).

An active aggregator (say av_g), upon receiving an event message e_q^r, behaves identically to a passive aggregator except in case 3. In that case, it first checks whether another active aggregator is already designated as the coordinator of e_q. If so, it just inserts e_q^r into its buffer as the first message of e_q. AA av_g will eventually finds out the notification cycle details from e_q's coordinator (if it does not know already). If av_g is not aware of any other node claiming the coordinator-hood of e_q, it executes the coordinator establishment protocol described in the next sub-section. In all three scenarios, if e_q^r was sent to av_g by a passive aggregator, av_g informs the passive aggregator about the coordinator and the notification cycle details of e_q.

The coordinator performs all the aggregation-related duties described above. In addition, at the end of every notification cycle, it receives partially aggregated messages from passive and active aggregators. These messages are merged and any redundancies are eliminated. The merged message is then sent to the subscribers.

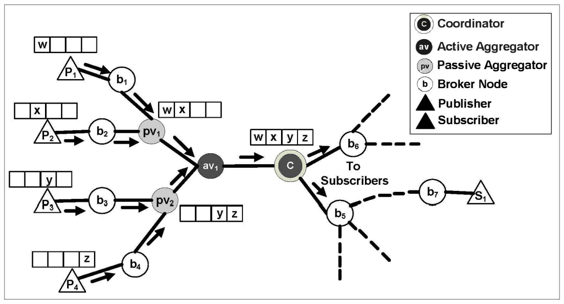


Fig. 1: Distributed Message Aggregation in *Caeva*

Figure 1 depicts the multi-stage merging at the passive/active aggregators and the coordinator.

Dynamic Coordination: When an active aggregator av_g receives a message of an event e_q with no established coordinator, av_g attempts to become the coordinator. It sends a message to all other active aggregators. An active aggregator av_h receiving such a message from av_g consents to av_g's claim if av_h has not attempted to become the coordinator of e_q. Ties are broken in decreasing order of broker ID; the "winner" sends a denial message to the "loser", who consents. Once the coordinator is established, it determines the duration of the notification cycle and the start time of the first cycle. The coordinator sends its identity and the notification cycle to the relevant set of aggregators; these aggregators in turn forward partially aggregated messages to the coordinator "just in time" (before the end of the notification cycle) to avoid additional latency.

Dynamic and Adaptive Aggregator Placement: We now describe our adaptive passive aggregator placement algorithm. This algorithm adapts the placement of the passive aggregators based upon the patterns of published event messages. This algorithm executes continuously in the background, and at the conclusion of each event, it decides whether to alter the positions of the passive aggregators or to maintain the current placement. When altering the PA placement, the PAs are moved by only one hop at each step. In other words, at the end of an event, the algorithm decides one of three things: (1) maintain the current PA placement; (2) move the PAs one hop away from the active aggregators (towards the edge of the routing AG); or (3) move the PAs one hop towards the center of the routing AG. The decision is based on the estimated costs and benefits of each option.

Three types of brokers are involved in executing the algorithm, namely, the current set of PAs, the immediate upstream brokers of the current PAs (parents of current PAs) and the active aggregator of the event under consideration. Each parent broker estimates the benefits and costs of moving the PA functionality from its children to itself (i.e., moving its downstream PAs one hop closer to the center), while each PA estimates the costs and benefits of moving the PA functionality to its children brokers (i.e., moving PAs one hop away from the center). The estimates from all PAs and parent brokers are consolidated at the active aggregator, which computes the cumulative costs and benefits of the three options and adapts the PA placement accordingly.

Now we discuss the formulations for estimating the costs and benefits for moving PAs one hop closer and one hop away from the center of the routing AG. First, we explain the cost and benefit formulae for moving PAs one hop closer to the center. Each parent broker uses these formulae to calculate the costs and benefits of moving PA functionality from its children to itself. Consider one such parent node pt_x. Let $CH(pt_x) = \{pv_1, pv_2, \ldots pv_Y\}$ be its children brokers (note that these nodes are a subset of the current $PvSet$). Let H denote the distance between the active aggregator and the current $PvSet$. For any broker b_i of the overlay, let $Pm(b_i)$ denote the number of messages of an individual event e_q published directly at b_i (i.e., published by publishers directly connected to b_i), $Fm(b_i)$ denote the number of messages of the same event forwarded by its downstream neighbors, and $Rm(b_i)$ represent the sum of $Pm(b_i)$ and $Fm(b_i)$. Let Nc denote the number of notification cycles for which the event e_q lasts ($Nc = \frac{dn(e_q)}{t_m}$, where $dn(e_q)$ denotes the total duration for which the messages pertaining to e_q are published and t_m denotes the length of the notification cycle.

We now formulate the benefits of moving the PA functionality from $\{pv_1, pv_2, \ldots pv_Y\}$ to pt_x. If pt_x were to assume the PA functionality, it would send one aggregated message to the coordinator at the end of each notification cycle instead of $pv_1, pv_2, \ldots pv_Y$ individually sending an aggregated message at the end of each notification cycle. Furthermore, the aggregated message from pt_x would need to travel one hop fewer than the messages from the aggregated messages from the current PAs. Thus, the number of message hops saved over the entire duration is $Nc \times (H \times Y - (H - 1))$. Also, if pt_x assumes the PA functionality, the messages published directly at pt_x would be aggregated/eliminated immediately, thereby avoiding the need for these messages to individually travel until the coordinator. Therefore the benefits of moving the PA functionality to pt_x is $BN(pt_x) = Nc \times (H \times Y -$

$(H-1)) + Pm(pt_x) \times (H-1)$. However, there are also costs associated with moving the PA functionality to pt_x. Notice that if pt_x becomes the PA, all the messages received at $pv_1, pv_2, \ldots pv_Y$ have to travel one extra hop before being aggregated. Therefore, the extra overheads involved in moving PA functionality to pt_x is $CN(pt_x) = \sum_{pv_y \in CH(pt_x)} Rm(pv_y)$. Thus, the relative savings obtained by moving the PA functionality to pt_x is $SN(pt_x) = BN(pt_x) - CN(pt_x)$.

Through a similar reasoning, we can compute the costs ($CF(pv_i)$) and benefits ($BF(pv_i)$) of moving the PA functionality from an arbitrary passive aggregator pv_i to its Z child brokers $\{cp_1, cp_2, \ldots, cp_Z\}$, respectively, as $CF(pv_i) = NC \times ((H+1) \times Z - H) + Pm(pv_i) \times H$ and $BF(pv_i) = Fm(pv_i)$. Thus, the savings obtained by transferring PA functionality to child brokers of pv_i is $SF(pv_i) = BF(pv_i) - CF(pv_i)$. Note that SN and SF can acquire negative values.

At the end of culmination of an event, the coordinator obtains the SF values from each current passive aggregator and SN values from each parent broker of current passive aggregators. It then sums up the various SN values to obtain the cumulative SN (CSN) value, and it computes the cumulative SF (CSF) value as the sum of various SF values. These values are used in adjusting the PA placement as follows. If $CSF \geq \delta$ then PAs are moved one hop away from the center. If on the other hand, $CSN \geq \delta$ then PAs are moved one hop closer to the center. If neither condition holds, then PAs are maintained at their current positions.

One issue that still need to be addressed is that of preventing thrashing (PAs continuously alternating between two positions). We achieve this by introducing an extra condition. The PA adaptation direction can be reversed only when the estimated savings are higher than the savings in the previous adaptation that brought PAs to their current position. Concretely, suppose in the last adaptation the PAs moved one hop closer to the center and the estimated cumulative savings (CSN) was μ. The PAs move back to their earlier positions (one hop away from the center) only if the estimated savings (CSF) of the current adaptation is higher than μ. Otherwise the PAs are maintained at their current positions even though $CSF \geq \delta$. An analogous strategy is adopted for moving the PAs closer to the center when they had moved away in the last adaptation.

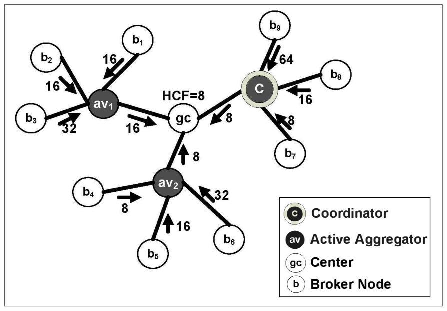
C. *Subscriber-Customized Notification Cycle*

Finally, we describe how *Caeva* allows each subscriber to choose its notification cycle duration. In the *Caeva* prototype, a subscriber can choose its notification cycle duration in integer multiples of minimum notification duration (md). As mentioned before, a client specifies this at subscription time. A simple and naive way of implementing a customized notification cycle would be to hoard the notification messages sent out by the coordinator at the broker that is directly connected to an arbitrary subscriber s_i. The broker would send out notification messages to s_i at appropriate instances of time. However, this leads to unnecessary messaging within the overlay.

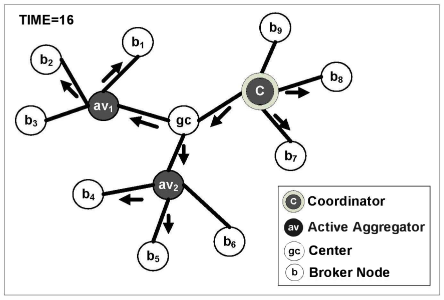
Instead, *Caeva* sends a notification through a path of the routing AG only if there is a subscriber downstream that should receive the notification at current instance. This is achieved by a combination of *upward propagation of subscriber preferences* and *selective downward dissemination of notifications*.

Upward Preference Propagation: The subscriber chooses its notification cycle duration in integer multiples of md. An arbitrary leaf broker of a routing AG, say b_k, may have multiple subscribers with different notification cycle durations. The edge broker calculates the highest common factor (HCF) of the notification cycle durations of the subscribers directly attached to it. This value indicates the period at which b_k should receive notification from its upstream node. Broker b_k sends this value to its upstream neighbor. A non-leaf broker, say b_j, calculates the HCF of the values sent by its downstream neighbors and the notification cycle durations of the subscribers directly attached to it, and propagates to its upstream neighbor. This is the period at which b_j should receive notification from its upstream neighbor. This process culminates at the graph center, which performs the same computation. The result is the HCF of the notification cycle durations of *all* subscribers being served by the routing AG. This value is maintained at the center and is used by the coordinator as the cycle duration for issuing aggregated messages. Figure 2(a) illustrates the upward preference propagation mechanism on a routing AG with 13 brokers. The HCF of the notification durations of all subscribers is 8, which is used as the cycle duration for issuing aggregated messages.

Selective Notification Dissemination: As described in Section III-B, at the end of each cycle the coordinator obtains partially aggregated messages from various aggregators and merges them to create a notification message. However, the aggregated message at the end of a particular cycle needs to be sent only if subscribers depend upon their notification cycle preferences. Thus, instead of blindly sending the aggregated message through the routing AG, the coordinator checks which of its neighbors should receive notification at the current



(a) Upward Preference Propagation



(b) Selective Notification Dissemination

Fig. 2: Illustration of Customized Notification Scheme

time and sends the aggregated message only to them. The intermediate brokers and the leaf brokers also work in a similar fashion. When a broker b_j receives an aggregated message from its upstream neighbor, it sends the message to only those downstream neighbors (if any) and subscribers (if any) that are due to receive the message at the current time. If the message is not sent to at least one downstream neighbor or subscriber, b_j maintains the message in a temporary buffer. While sending a message to a downstream broker, say b_k, b_j sends all those fields that have not been sent to b_k but are available currently at b_j. The exact same process is followed when sending messages to subscribers. Figure 2(b) demonstrates the selective notification dissemination technique at time 16. Notice that av_2 sends the aggregated message to b_4 and b_5, but not to b_6.

IV. Experimental Results

Caeva has been implemented on top of the *Siena* pub-sub infrastructure [3]. We have performed several experiments to study the performance of *Caeva*.

A. Setup

Our experiments were set up as follows. In all cases we use a random graph topology. Each complete event in our experiments consists of 20 fields, including the event key. In published messages, the number of fields that holds valid data varies from 1 to 10. The number of messages pertaining to an individual event can vary, and they are generated in the following manner. Each publisher of a particular event generates messages pertaining to that event according to a Poisson process. The event duration is chosen to be a maximum of 100 time units. In our experiments, all nodes subscribe once and for any event. The particular event and associated field names are selected according to a uniform random distribution.

In our experiments, we use a merge threshold (denoted T_m) and a redundancy threshold (denoted T_r, and this value is fixed in our experiments). T_m is the notification cycle (defined in the Section III-C), T_r is the amount of time messages are buffered at broker nodes in an attempt to discard later redundant messages.

Overall, an experiment is defined by its spatial locality for publishers, redundancy ratio for messages, and values for T_m and T_r. Spatial locality can be defined using the median distance between all pairs of publishers. However, in practice, it is difficult to set these distances in *Caeva* (due to limitations in *Siena*). Therefore, we vary the spatial locality between three configurations: (1) completely local, where all publishers reside at the same point in the graph; (2) partially local, where there are a few clusters of publishers, and (3) non-local, where all publishers are at different points in the graph.

In addition, the messages sent by the publishers for a given event can vary in their redundancy. We define the *redundancy ratio* for an event as F_r/M_t, where F_r denotes the number of messages whose fields are a subset of the fields previously sent, and M_t is the total number of messages sent. In our experiments, both T_m and T_r ranged between 0 and 10 simulated time units, such that $T_m \leq T_r$.

In the experiments below, we generally measure three different implementations. *Siena* provides the baseline. *Agele* is our previous system [12], on which *Caeva* is based; *Agele* is centralized, static, and uses one center node for aggregation, while *Caeva* is distributed and adaptive. Generally, we examine three important metrics: (1) percentage of the messages that are suppressed (by merging or duplicate elimination), (2) extra time that is added due to buffering at aggregators (measured by when the complete event is received), and (3) complete events and amount of data that subscribers receive.

B. Effect on Broker Overlay

We begin by investigating the effect that *Siena*, *Agele*, and *Caeva* have on the broker overlay. Here, we are interested in the total messages in the system. For this experiment, we use a random topology, low spatial

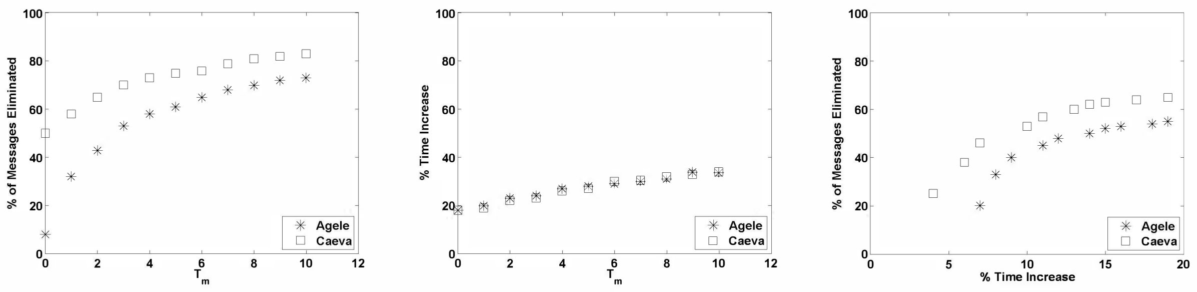


Fig. 3: When T_m varies, percentage of messages in broker overlay suppressed (left); time increase (center). On the right, tradeoff between delay and percentage of messages eliminated.

locality, and the medium redundancy ratio. For *Agele* and *Caeva*, we vary T_m in the experiments. All results are relative to *Siena*.

Figure 3 shows the results. Because *Siena* does not handle redundant and partial event messages, it incurs more messages than either *Agele* or *Caeva*. In particular, *Caeva* eliminates up to 80% of the messages in the overlay. Comparing *Caeva* to *Agele* shows that the former suppresses more messages as T_m increases. This is because *Caeva* eliminates messages at the passive aggregators, which are closer to the publisher. This has two beneficial effects: (1) it takes additional message load off of broker nodes in between the passive aggregators and the coordinator, and (2) it can, in some situations, take additional message load off of brokers in between the coordinator and the subscribers. The latter point is somewhat subtle: if a message is not eliminated at the passive aggregator, then it proceeds to the coordinator. The coordinator may eliminate it, but it is possible that T_m is sufficiently small that it is *not* eliminated.

The center graph in the figure shows a time increase (for completed events) for both *Caeva* and *Agele*. Additionally, as expected, the relative time increase is larger with larger T_m. One item to note is that *Caeva* and *Agele* have essentially the same overhead. This is by design—the passive aggregators flush their buffered messages such that they reach the coordinator just in time to be flushed to the subscriber. (The small difference is because the coordinator in *Caeva* is a different broker node than the center in *Agele*.) The right graph shows similar information to the left and center graphs, but specifically shows the tradeoff between increased latency and the number of messages eliminated.

Next, we study the effect on the broker overlay when the spatial locality of the publishers as well as the redundancy ratio vary. We used the spatial localities and redundancy ratios specified above. In the graph, the first letter refers to the spatial locality; "H" for completely local, "M" for partially local, and "L" for non-local. The second letter refers to the redundancy ratio; "H" for a redundancy ratio of 85%, "M" for 50%, and "L" for 20%. In these tests, T_m and T_r are both 10. Figure 4 shows the results. We see that as the spatial locality of the publishers increases, the advantage of *Caeva* increases over *Agele*, in terms of message load in the broker overlay. This is because more of the published messages are directed to the same passive aggregator, which eliminates some of them.

Publishers	**Static**		**Adaptive**
	Min	Max	
3	101,796	125,714	102,583
7	147,913	220,126	150,239
31	181,189	232,420	197,141
255	203,241	227,747	211,375

TABLE I: Number of messages for different numbers of publishers for both static and adaptive algorithm

We note that many scenarios of publisher locality and redundancy ratio are possible. For example, a news bulletin occurring at night would potentially lead to widely distributed publishers, whereas an accident during rush hour would likely lead to mostly localized publishers. *Caeva* is actually the best choice for all of these cases, though its advantage increases with more locality in space and time. The one disadvantage of *Caeva* relative to *Agele* is that it is more complex and involves more broker-broker communication.

C. Adaptive PA Placement

Table I shows the number of messages for different numbers of publishers for both the static and adaptive algorithms. For the static algorithm, the passive aggregators can reside at several different places; we show both the minimum and the maximum. This experiment uses publishers with similar characteristics. The key point is

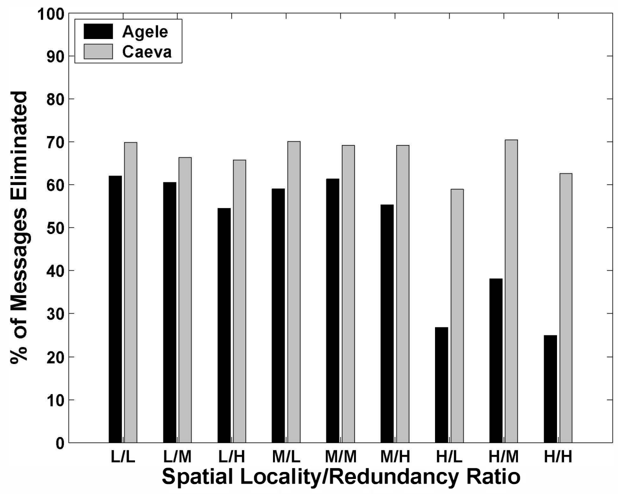


Fig. 4: Percentage of messages in broker overlay suppressed when spatial locality and redundancy ratio vary; the first letter indicates the locality, and the second the redundancy ratio

Varying Publishers	**Static**		**Adaptive**
	Min	Max	
Uniform	153,847	203,474	153,385
Nonuniform	293,265	361,287	266,722

TABLE II: Number of messages for different numbers of publishers for both static and adaptive algorithm when publishers have nonuniform characteristics

that the adaptive algorithm is always close to as good as the minimum and avoids the large penalty of choosing the maximum. Keep in mind that the static algorithm requires a single placement, and without application-specific knowledge, it is possible that a bad placement might be chosen.

Next, Table II shows the same attributes, but compares the uniform and nonuniform publisher case. It is clear that for nonuniform publishers, the adaptive algorithm is significantly (10%) better. This is because when publisher characteristics change, the static algorithm cannot change. On the other hand, the adaptive algorithm changes based on these characteristics.

V. Related Work

Over the past decade, various aspects of pub-sub systems have been widely studied including subscription mechanisms, architectures, quality-of-service, mobility, and reliability [6], [14], [2], [3], [15], [16], [17], [9], [18], [19], [7], [20]. Surprisingly, the issue of redundant and partial event messages, which are very common in settings with human participants, has received little research attention. A few researchers have considered the problem of *exact* duplicate elimination [20], [21], [22]. However, most solutions are simplistic with performing duplicate elimination at the subscribers being the most common approach [20]. The XTreeNet system [21] uses an in-network duplicate elimination scheme. However, this technique is not effective in reducing message traffic due to duplicates originating from different regions of the overlay. In addition, an event-message is cached at each node in its path from publisher to subscribers with very little coordination among these nodes. Thus, the system is not able to provide any guarantees to the subscribers or offer them flexibility with respect to the degree of duplicate elimination or the notification times. To the best our knowledge, our previous work with *Agele* [12] was the first system to consider incomplete (partial) event messages aggregation. *Agele* is a centralized system that uses a center node to aggregate all messages; i.e. there is one, fixed active aggregator and no passive aggregators. In addition, *Agele* is static; the notification cycle is fixed over the entire system. *Caeva* is much different; it is distributed and therefore scalable, it allows flexible, adaptive placement of passive aggregators as well as a flexible choice of the notification cycle for each subscriber.

The area of distributed stream processing [23], [24], [25], [26], [13] has similarities to event aggregation in decentralized pub-sub systems. In both these cases, data originating from the nodes of an overlay needs to be processed and delivered to a set of recipient nodes. However, there are also crucial differences between the two. First, in stream processing systems, the source nodes of various data streams are generally known when the query plan is evolved. Second, the data streams last for relatively long durations of time, and so do the data processing operators defined on these streams. Third, many of the stream processing systems assume a global view of the overlay topology. These characteristics justify and permit the heavy-weight, optimization-based query planning, operator placement, and adjustment strategies used by stream processing applications. The pub-sub environment, especially in community-oriented applications, is much more ad-hoc — publishers generate event messages in a non-continuous manner and at arbitrary points in time. Furthermore, each event is active for short duration of time, in the sense that the messages pertaining an event are published in a short time window. Thus, the heavy-weight operator placement strategies are not appropriate for *Caeva*.

Complex event detection [23] also bears similarities to event aggregation. However, most of the current approaches to complex event detection rely upon a priori planning which assumes that the event sources are known before hand.

In contrast to these systems, *Caeva* does not require a priori knowledge of event message sources, and its protocols and techniques are lightweight and dynamic.

VI. Conclusion

The pub-sub substrates of many community-oriented applications are faced with event streams that have various kinds of noise, including partial and redundant event messages. Effective handling of of this kind of noise is critical to the success of these applications; yet, it is challenging, especially in decentralized pub-sub systems.

The work presented in this paper describes the design and implementation of *Caeva*, which is a decentralized, scalable system for eliminating redundant and partial event messages. Scalability is achieved by aggregating events at multiple broker nodes, as the event messages are propagated from publishers to subscribers. In addition, *Caeva* has flexible and adaptive algorithms for placing aggregators and choosing notification cycles for subscribers. Results showed that *Caeva* is effective in terms of eliminating messages, limiting the increase in event latency, and adapting to changing event publication patterns.

Acknowledgment

This work was partially supported by the NSF grant CNS-0716357. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] R. Baldoni, C. Marchetti, A. Virgillito, and R. Vitenberg, "Content-based Publish-Subscribe over Structured Overlay Networks," in *Proceedings ICDCS*, 2005.

[2] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. E. Strom, and D. C. Sturman., "An Efficient Multicast Protocol for Content-Based Publish-Subscribe Systems," in *ICDCS*, 1999.

[3] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf, "Design and evaluation of a wide-area event notification service," *ACM Transactions on Computer Systems*, vol. 19, no. 3, pp. 332–383, 2001.

[4] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron, "SCRIBE: A Large-Scale and Decentralised Application-level Multicast Infrastructure," *IEEE Journal on Selected Areas in Communications (JSAC)*, 2002.

[5] B. Segall, D. Arnold, J. Boot, M. Henderson, and T. Phelps, "Content Based Routing with Elvin4," in *AUUG2k*, 2000.

[6] "TIB/Rendezvous," White paper, 1999.

[7] P. Pietzuch and J. Bacon, "Hermes: A Distributed Event-Based Middleware Architecture," in *Proceedings DEBS*, 2002.

[8] S. Voulgaris, E. Riviere, A.-M. Kermarrec, and M. van Steen, "Sub-2-Sub: Self-Organizing Content-Based Publish Subscribe for Dynamic Large Scale Collaborative Networks," in *Proceedings of IPTPS*, Feb 2006.

[9] P. T. P. Felber, R. Guerraoui, and A.-M. Kermarrec, "The Many Faces of Publish/Subscribe," *ACM Computing Surveys*, vol. 35, no. 2, 2003.

[10] "Digg (http://digg.com)."

[11] "Twitter (http://twitter.com)."

[12] J. Chen, L. Ramaswamy, and D. K. Lowenthal, "Towards efficient event aggregation in a decentralized publish-subscribe system," in *Proceedings of DEBS*, 2009.

[13] P. R. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. I. Seltzer, "Network-Aware Operator Placement for Stream-Processing Systems," in *Proceedings of ICDE*, 2006.

[14] I. Aekaterinidis and P. Triantafillou, "PastryStrings: A Comprehensive Content-Based Publish/Subscribe DHT Network," in *ICDCS*, 2006.

[15] B. Chandramouli, J. M. Phillips, and J. Yang, "Value-Based Notification Conditions in Large-Scale Publish/Subscribe Systems," in *Proceedings of VLDB*, 2007.

[16] P. Costa, M. Migliavacca, G. P. Picco, and G. Cugola, "Epidemic Algorithms for Reliable Content-Based Publish-Subscribe: An Evaluation," in *Proceedings of ICDCS*, 2004.

[17] G. Cugola and L. Mottola, "A Self-Repairing Tree Overlay Enabling Content-based Routing in Mobile Ad Hoc Networks," *IEEE Transactions on Mobile Computing*, 2008.

[18] G. Li, S. Hou, and H.-A. Jacobsen, "A Unified Approach to Routing, Covering and Merging in Publish/Subscribe Systems Based on Modified Binary Decision Diagrams," in *ICDCS*, 2005.

[19] J. P. Loyall, M. Gillen, and P. Sharma, "QoS Allocation Algorithms for Publish-Subscribe Information Space Middleware," in *MIDDLEWARE*, 2008.

[20] Y. Huang and H. Garcia-Molina, "Publish/subscribe in a mobile environment," *Wireless Networks*, vol. 10, no. 6, 2004.

[21] W. Fenner, M. Rabinovich, K. K. Ramakrishnan, D. Srivastava, and Y. Zhang, "XTreeNet: scalable overlay networks for XML content dissemination and querying," in *Proceedings WCW*, 2005.

[22] M. Srivatsa and L. Liu, "Securing Publish-Subscribe Overlay Services With EventGuard," in *Proceedings of ACM-CCS*, 2005.

[23] M. Akdere, U. Çetintemel, and N. Tatbul, "Plan-based complex event detection across distributed sources," in *Proceedings of VLDB*, 2008.

[24] M. Branson, F. Douglis, B. Fawcett, Z. Liu, A. Riabov, and F. Ye, "CLASP: Collaborating, Autonomous Stream Processing Systems," in *Proceedings of MIDDLEWARE*, 2007.

[25] B. Chandramouli and J. Yang, "End-to-End Support for Joins in Large-Scale Publish/Subscribe Systems," in *Proceedings of VLDB*, 2008.

[26] V. Kumar, B. F. Cooper, Z. Cai, G. Eisenhauer, and K. Schwan, "Resource-Aware Distributed Stream Management Using Dynamic Overlays," in *ICDCS*, 2005.