Towards Efficient Event Aggregation in a Decentralized Publish-Subscribe System

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

David Lowenthal
Department of Computer Science
The University of Arizona
dkl@cs.arizona.edu

ABSTRACT

Recently, decentralized publish-subscribe (pub-sub) systems have gained popularity as a scalable asynchronous messaging paradigm over wide-area networks. Most existing pub-sub systems, however, have been designed with the implicit assumption that published data is clean and accurate. As the pub-sub paradigm is incorporated in real-world applications with human participants, this assumption becomes increasingly invalid due to the inherent noise in the event stream. The noise can take many forms, including redundant, incomplete, inaccurate, and even malicious event messages.

This paper explores the distributed computing issues involved in handling event streams with redundant and incomplete messages. Given a distributed broker overlay-based pub-sub system, we present our initial ideas for (1) aggregating event information scattered across multiple messages generated by different publishers and (2) eliminating redundant event messages. Key to our approach is the concept of an event-gatherer—a designated broker in the routing graph that acts as a proxy sink for all messages of a particular event—located at the graph center of the corresponding routing tree. This paper proposes a novel decentralized algorithm to find this graph center. Early results show that the proposed scheme typically reduces the message load by over 60% with less than 25% time overhead to subscribers.

1. INTRODUCTION

A decade of research and development has established the publish-subscribe (pub-sub) framework as a distinct and an important data communication paradigm [7, 8, 10, 12, 30, 3, 7, 27, 36, 17]. It is loosely coupled and uses an asynchronous communication model, which makes it particularly well-suited for large-scale distributed applications. The pub-sub paradigm is increasingly being incorporated into community-oriented applications, which are characterized by human participants. Examples include online social networks such as Digg [1] and Twitter [2].

Most existing pub-sub systems make an implicit assumption that the published event data is clean, complete, and accurate (i.e., the event data is devoid of noise). While the assumption may be valid in applications where the publications are electronically generated (i.e, the publishers are gadgets such as servers and different kinds of sensors), this is hardly the case in pub-sub systems with human participants. In community/social group-oriented event services noisy events are almost a given. This noise can take various forms, such as redundant events, incomplete event messages, inaccurate event messages, and even events generated with malicious intent.

Redundant event messages and messages containing incomplete (partial) information about the corresponding events are among the most common forms of noise in applications with human participants. Ideally, superfluous messages should be filtered by eliminating redundant event messages and aggregating event messages containing partial information. However, doing so poses significant challenges both from data management and distributed systems perspectives. This paper is devoted to the study of the distributed computing issues involved in effectively dealing with redundant and partial events in pub-sub systems based upon decentralized broker overlays[1].

In this paper, we present *Agele*, a pub-sub system that embodies our initial ideas towards reducing superfluous messages from potentially distinct publishers in semi-real time. *Agele*'s architecture is based upon a distributed broker overlay, where the message brokers interact with one another in peer-to-peer (p2p) fashion. The twin design goals of *Agele* are to minimize the message load in the overlay and to simultaneously minimize the latency overheads of subscribers. These goals are achieved by elimination and aggregation of event messages as they travel through the overlay from their publishers to the subscribers.

Specifically, this paper makes three unique contributions.

- First, we introduce the concept of *event gatherers*, which are broker nodes that identify and eliminate redundant event messages as well as temporarily hold and merge partial event messages. We show that in order to achieve our goal of minimizing message load in the system, the event gatherer for a set of related subscription predicates should be located at the *graph center* of the corresponding routing acyclic graph.
- Second, we present a novel decentralized algorithm for determining the graph center of a acyclic graph. An important feature of our algorithm is that it does not need a global view of the overlay, and it only relies upon message exchanges between neighboring nodes in the overlay. Further, our algorithm is efficient and provably accurate.
- Third, we perform experiments to study the viability of our

[1]Data management issues are also important for the ultimate realization of the system. But, they are beyond the scope of the current paper

DEBS'09, July 6-9, Nashville, TN, USA.

ideas. Early results are encouraging in the sense that *Agele* reduces the message load by over 90% in some cases and over 60% in most. Furthermore, the overhead to subscribers can be kept typically to around 25%, and results in a real-world system with stringent bandwidth constraints would likely be much smaller.

The rest of this paper is organized as follows. Section 2 discusses the motivation behind our work. Then, we provide an overview of our architecture in Section 3. Next, Section 4 presents our decentralized center-finding algorithm, and then we discuss the implementation of *Agele* in Section 5. The results are presented in Section 6. Finally, Section 7 describes related work, and Section 8 summarizes the paper and discusses our plans for future work.

2. MOTIVATION AND CHALLENGES

Recently, community/social group-oriented event services such as twitter [2] are gaining popularity. In these applications, the members of a (possibly ad-hoc) community or social group collaboratively report and receive events that may be of interest to that group. Observe that this is a pub-sub system in which participants (both publishers and subscribers) are human. Generally, the application provides a subscription mechanism through which a participant can specify the events of interest. Participants who notice a particular event that may be of interest to the community report it to the system using their communication device (desktops, laptops, mobile gadgets, etc.).

In such applications, several participants may notice an event simultaneously (or within a short duration of time). However, some or all of these participants may only have partial information about the event. The event messages published by these participants would naturally contain partial information. Also, individual messages may contain varying degrees of information about the event. Since several participants may individually publish an event to the system, it can lead to redundant event messages. Event message redundancy can be of two forms. First, two event messages may contain the exact same information. This *direct* form of redundancy is generally referred to as *exact duplicates*. However, there is another, subtler form of redundancy. The information contained in an event message might have already been provided in "bits and pieces" by a set of publishers. In other words, a set of previous messages *cumulatively contains* all the information that the current message is carrying. We call this *indirect* redundancy. As an example, consider the example of a collaborative traffic incident report system, wherein participants report traffic incidents that they notice, which would then be delivered to set of subscribers. In such a system, many participants might report a traffic incident. While all messages pertaining to an accident would contain its location, some of them might provide details about the accident such as how many cars are involved, whether is a fire and whether there are casualties. Some may contain information that is already known from earlier messages.

The simplest strategy for dealing with redundant and partial event messages would be to relegate these responsibilities to the subscribers. This *hands-off* approach, however, is fraught with several drawbacks. First, the subscriber devices may not have the computational and communication capabilities to deal with redundant and partial event messages. For each actual event, subscribers may receive several messages that may overwhelm low-end devices, thus causing them to drop legitimate and important messages. Second, even if the devices were capable of handling the message load, this approach would lead to significant bandwidth overhead, and, in case of mobile devices, corresponding battery-power drain. Third, it would also lead to high communication overhead within the pub-sub system, thereby affecting its scalability and efficiency. Fourth, this approach assumes that the subscriber devices have the software necessary for for performing aggregation and elimination, which raises practicality concerns due to the vast heterogeneity among the subscriber devices.

Thus, it is desirable to have an underlying system that aggregates event information and eliminates redundant messages so that a single (or at most a few) consolidated event messages are delivered to the subscribers.

2.1 Research Challenges

Designing a distributed broker network that delivers consolidated non-redundant event messages to the subscribers poses a number of distributed systems problems. First, since the messages corresponding to an event may be published by several publishers, event messages may enter the broker overlay through many different brokers, and, depending upon the routing scheme employed by the system, the event messages might follow non-overlapping (or barely overlapping) paths to the subscribers. Hence, individual brokers in the overlay might not even be aware of the existence of multiple (redundant and incomplete) event messages. Second, the individual messages pertaining to an events might be published over a certain period of time, which necessitates storage and maintenance of aggregated information about different events. The questions in this regard are where (on which broker node(s)) would this information be maintained and for how long? Third, consolidating messages pertaining to an event implies that there would be an inherent delay in notifying the subscribers of the information that has been published about it. Managing the tradeoffs between the extent of aggregation and the delay in communicating published information about events to the respective subscribers is another challenge. Fourth, failure of individual brokers may result in loss of aggregated event information in addition to service disruptions. Protecting loss of aggregated event information in the face of broker node failures is yet another problem.

3. ARCHITECTURE OVERVIEW

In this section, we provide an architectural overview of our system. Our event aggregation and redundancy elimination model is generic, and it can be applied to most pub-sub frameworks. However, for conceptual clarity, in this paper, we focus on a framework that is similar to type-and attribute-based pub-sub paradigm [27] (details about the subscription model is provided later in the section). However, the proposed architecture as well as the associated techniques can be adapted to topic-based or content-based publish-subscribe systems.

Agele is based upon a distributed overlay of message brokers (also referred to as *nodes*). The N message brokers in the overlay are 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. 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. Each publisher and subscriber must be connected to one of the brokers.

3.1 Events, Messages and Subscriptions

Similar to type-and attribute-based pub-sub paradigm [27], every event in our system is associated with a *topic*, which provides a broad context for the event. Continuing with the example presented in the previous section, 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*. Within a certain time-window, the key along with the topic corresponds uniquely (or can be resolved uniquely) to an event. In our system, the key field is descriptive, and it can be used in subscription predicates. For example, the key for a traffic incident event would be the street intersection at which it occurs. The number of fields of an event, their types and the key are determined by the event's topic.

However, in contrast to existing pub-sub systems, there can be multiple messages associated with a single event, and these messages may have been published by multiple publishers. The messages corresponding to an event e_q are represented as $\{e_q^1, e_q^2, \ldots, e_q^U\}$. Each message provides some (possibly partial) information about the event. The fields of an event message e_q^r are represented as $\{e_q^r(1), e_q^r(2), \ldots, e_q^r(V)\}$. In a message that carries partial information about the event, one or more fields (other than the event key) would be empty. Our assumption of key-topic uniqueness implies that if the first message of an event is published at time t_f, any messages with the same key-topic pair generated between t_f and $t_f + W$ correspond to the same event.

In *Agele*, 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. *Advertisements* are messages used by publishers to indicate 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. Notice that the basic *Agele* framework can be considered as a special kind of content-based pub-sub model, where the content of every published item includes a topic and an event key. In fact, the *Agele* implementation is based on a content-based pub-sub system, Siena [10].

3.2 Routing

As in many pub-sub systems [8, 10], an acyclic network of brokers, called a *routing acyclic graph (routing AG, for short)* forms the basis for routing events from publishers to consumers. Routing AGs are embedded in the broker overlay. Notice that routing AGs are connected, non-directed, and acyclic (may also be considered as non-rooted trees). Similar to *Siena* [10] and *Hermes* [27], routing AGs in *Agele* are constructed in a completely decentralized fashion by peer-to-peer forwarding of subscriptions and advertisements. Furthermore, the predicates of subscriptions with the *same* topic are aggregated at brokers, and a more generic subscription is forwarded. In effect, *Agele* maintains a distinct routing AG for each topic. However, individual brokers can belong to multiple routing AGs. When available, advertisements can be utilized for optimizing the routing AGs.

3.3 Event Gatherers

We now explain our mechanisms for message aggregation and redundancy elimination. Our strategy is to merge partial messages and eliminate redundant messages within the routing AG as they travel from their publishers to the subscribers. One of the nodes in the routing AG is designated as the *event-gatherer* for events disseminated through that routing AG. All the event messages with the same topic value are routed through the corresponding event gatherer. The event gatherer, upon receiving an event message, determines whether (1) this is the first message in a possible sequence of messages from the same event, in which case it is stored, (2) it is redundant, and hence needs to be eliminated, or (3) it can be merged with an existing message. At an appropriate point in time (see below), the event-gatherer disseminates the aggregated message to the respective subscribers.

Concretely, an event-gatherer maintains a message buffer, which stores at most one message per distinct event. When a message comes in, the event gatherer checks whether an event message with the same event key already exists in the buffer. If so, there is an earlier message in the buffer corresponding to the same event. The event gatherer now determines if the newly arrived message is redundant (data contained in it is a subset of the data already available message existing in the buffer) or whether it can be merged with the existing message. Redundant messages are discarded. A message is *mergeable* if it contains some extra information that the existing message does not have (i.e., there exists one field which is empty in the existing message, but the corresponding field in the incoming message contains data). If the incoming message is deemed mergeable, the event gatherer creates a new aggregate message by combining the information available in the two messages and stores it in the buffer instead of the current buffered message. If the buffer does not contain an event with a matching key, the incoming message is inserted into the buffer.

An event gatherer is allowed to maintain the message corresponding to an event for at most W time units, since beyond this time another distinct event with the same event key might occur From message aggregation and redundancy elimination perspectives, it is optimal to have the event gatherers maintain the message corresponding for W time units, and after that send out the merged message to the actual recipients. However, waiting this long may sometimes affect the subscribers' ability to react properly to the event. To address this concern, the event gatherers periodically send out merged messages, while maintaining them in its buffer[2]. Specifically, for each event message in its buffer, the event gatherer evaluates the respective subscription predicates every T_m time units and sends out the merged message to a subscriber if its predicate is satisfied. However, subscription predicates might depend upon fields of the message that are currently *empty*. This issue can be resolved in two ways - the event gatherer might simply wait until all the fields necessary for evaluating the respective predicate are available, or it might *optimistically* treat the predicate terms for which the values are not available as *true*. In the later case, there is a chance of *false notifications*.

Figure 1 provides an example of the idea of eliminating messages with an event gatherer (labeled *Cen* in the graph). On the left, the center node buffers a message with field *a*, while messages containing fields *b*, *c*, and *d* are traveling through the graph. On the right, the latter three fields pass through the center node and are merged into one message. Moreover, the new, merged message containing all four fields is buffered in case redundant messages later arrive at the center node.

3.4 Where Should an Event Gatherer be Located?

Next, we explore the crucial question of *where an event gatherer should be located?* In other words, *on which brokers of the routing AG should the event gathering functionality be placed?* The location of the event gatherer within the routing AG has a significant impact on the message load in the broker overlay. Our main objective when choosing the event gatherer location is to minimize the message load in the broker overlay. A second, but equally important, objective is to minimize the latency overheads on the subscribers.

We now formulate the problem in terms of the first objective. However, as explained in our technical report [13], our solution

[2]Depending upon its buffer-space constraints, an event gatherer may store messages for shorter durations than what is permissible.

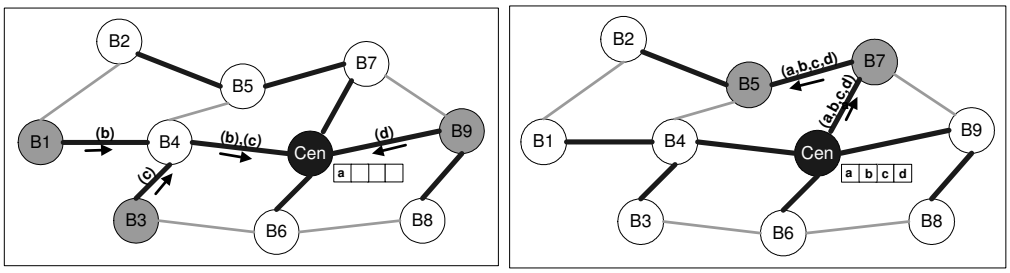


Figure 1: Pictorial representation of a topology with the center node indicated. All messages are part of the same event.

achieves both objectives. In the routing AG consisting of brokers $\{b_1, b_2, \ldots, b_M\}$, suppose the broker b_i is designated as the event gatherer. Now, if an event-message were to be generated by a publisher attached to node b_j, the messaging cost of transmitting this message to the event gatherer b_i is $Dist(b_j, b_i)$, where $Dist(b_j, b_i)$ represents the path length between the brokers b_j and b_i in terms of number of hops. Similarly, the cost of sending a merged message from the center to a subscriber b_k is $Dist(b_k, b_i)$. Since the event can enter into the system through any broker, the expected message load due to an event message is
$ECost(Rag, b_i) = \frac{1}{M} \sum_{b_j \in Rag} Dist(b_j, b_i)$,
where Rag represents the routing acyclic graph. Thus, the problem is to find a broker b_i such that $\sum_{b_j \in Rag} Dist(b_j, b_i)$ is minimized.

Traditional optimization strategies cannot be applied here as most of them are *centralized* schemes requiring a global view of the routing AG topology, whereas our system is based on a completely decentralized architecture wherein the brokers only interact with their neighbors. Furthermore, these strategies are computationally expensive. Therefore, we seek to develop an alternate solution that is not only amenable to decentralized implementation but is also efficient.

Our strategy is dependent upon the following observation. Since the cost of transferring a message from an arbitrary node b_j to the event gatherer b_i is determined by $Dist(b_j, b_i)$, in order to achieve low message loads, the event-gatherer should be located not too far away from any node in the routing AG. So, which broker in the routing AG satisfies this criterion? Intuitively, the event gatherer should be located in the *core region* of the routing AG. In fact, it can shown that the broker that satisfies this property would be the *graph center* of the routing AG. The graph center is defined as follows. Consider a undirected connected graph $G = (B, E)$ with nodes $B = \{b_1, b_2, \ldots, b_n\}$. The *distance* between two nodes b_j and b_i, represented as $Dist(b_j, b_i)$ is the length of the shortest path between them. The *eccentricity* of a node b_i is defined as the largest of the shortest paths between b_i and any other node in the graph. That is $Eccentricity(b_i) = Max_{b_j \in B}(Dist(b_i, b_j))$. The graph centers of G are the set of nodes with minimum eccentricity [3]. Since, graph center has the minimal longest path, it naturally satisfies the above criterion.

4. GRAPH CENTERS

The problem now is to develop a completely decentralized algorithm to determine the center of acyclic graph, which would execute only through message exchanges among neighboring brokers. Several algorithms have been proposed for finding graph centers [37, 35]. However, very few of them are applicable to a decentralized setting where a global view of the network is not available. To the best of our knowledge there are very few distributed algorithms for locating centers of general graphs [31, 34]. However, even these algorithms have significant limitations, which prevents us from utilizing them (see Section 7).

Before presenting our solution, we define a few concepts that are necessary for its description. Consider two neighboring brokers b_i and b_j in the routing AG. Since the routing graph is acyclic (and undirected) , if the edge (b_i, b_j) is removed, the graph is partitioned. The partition (subgraph) that contains the node b_j is called the *subgraph anchored by* (b_i, b_j) and is represented as $SG(b_i, b_j)$. Similarly, the partition that contains b_i is the *subgraph anchored by* (b_j, b_i) ($SG(b_j, b_i)$). Note that in the routing AG, the path from b_i to any node in $SG(b_i, b_j)$ has to necessarily pass through b_j. The subgraph eccentricity of b_i with respect to b_j (represented as $SEcc(b_i, b_j)$) is defined as the eccentricity of b_i with respect to only the nodes in $SG(b_i, b_j)$. In other words, $SEcc(b_i, b_j) = Max_{b_x \in SG(b_i, b_j)}(Dist(b_i, b_x))$. The neighboring brokers of b_i in the routing AG is represented as $NbrList(b_i)$.

4.1 Distributed Center Determination

As we remarked earlier, our algorithm is completely decentralized, and it relies purely on message exchanges among neighbor brokers. The algorithm works in three stages, as explained below.

4.1.1 Initiation Phase

In this phase any broker in the routing AG initiates the center determination algorithm. It does so by sending an initiation (INIT) message to all its neighbors in the routing AG. The INIT message contains the routing AG topic and the identifier of the broker initiating the process. Each broker receiving the INIT message propagates it to all its neighbors in the routing AG. Concurrent initiations are resolved by ignoring all but the first message.

4.1.2 Eccentricity Determination Phase

In the second phase, the brokers in the routing AG progressively discover their eccentricities with respect to the routing AG. A broker b_j sends one or more messages to its neighbor broker b_i, each of which updates the current value of $SEcc(b_i, b_j)$. Finally, when b_j discovers that current value of $SEcc(b_i, b_j)$ has reached its final value, it sends a STABLE message to b_i.

Concretely, the algorithm works as follows. Suppose b_i and b_j are neighboring brokers in the routing AG, and suppose $NbrList(b_i) = \{b_e, \ldots, b_h, b_j\}$ and $NbrList(b_j) = \{b_i, b_k, \ldots, b_p\}$. Every broker in the routing AG maintains its subgraph eccentricity with respect to each of its neighbors. Further, it also maintains the subgraph eccentricities of each of its neighbors with respect to itself. For example, broker b_i maintains $SEcc(b_i, b_x)$ and $SEcc(b_x, b_i)$ $\forall b_x \in NbrList(b_i)$. When a broker receives an initiation message, it initializes its subgraph eccentricities with respect to each of its neighbors to zero.

[3] There could be multiple graph centers, all of which have minimum eccentricities

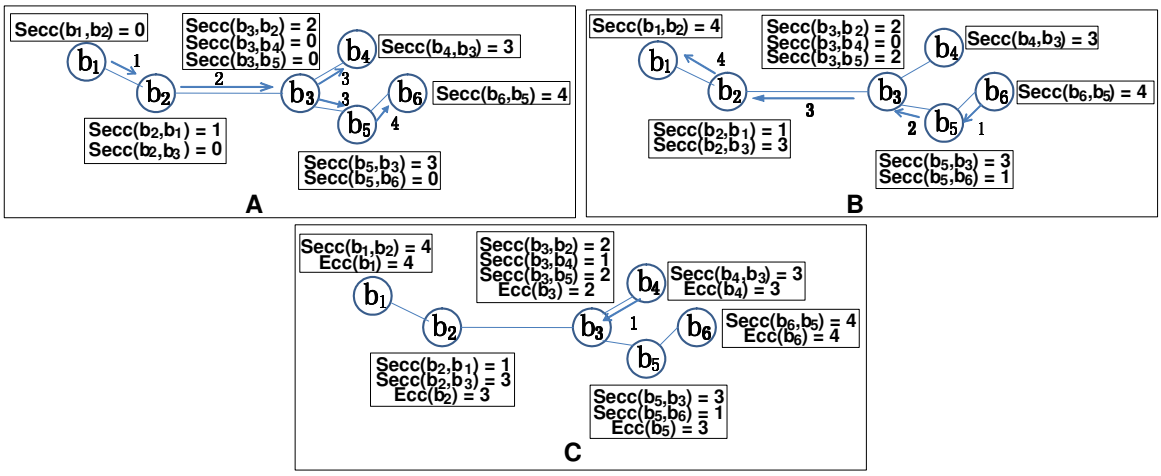


Figure 2: Pictorial representation of the eccentricity determination phase.

When a leaf broker, denoted b_k, receives an INIT message from its only neighbor (b_j), it sends back a UPDATE message containing $SEcc(b_j, b_k)$, which is equal to 1 (as b_k is the only node in the subgraph reachable through itself). It also updates its local copy of $SEcc(b_j, b_k)$ to 1. Further, b_k sends a STABLE message to b_j indicating that $SEcc(b_j, b_k)$ has stabilized. When an intermediate node b_j receives an UPDATE message from its neighbor b_k, it updates its copy of $SEcc(b_j, b_k)$. Then, for each of its neighbor brokers, b_j checks whether its subgraph eccentricity with respect to b_j needs to be updated. $SEcc(b_i, b_j)$ needs to be updated if the current value of $SEcc(b_i, b_j)$ is less than $SEcc(b_j, b_k) + 1$. If $SEcc(b_i, b_j)$ needs to be updated, b_j sets its local copy of $SEcc(b_i, b_j)$ to $(SEcc(b_j, b_k) + 1)$, and sends the same value as an UPDATE message to b_i.

Notice that $SEcc(b_i, b_j)$ can only updated when b_j receives an UPDATE message from one of its neighbors other than b_i. Therefore, if b_j receives a STABLE message from all of its neighbors other than b_i, it can safely conclude that $SEcc(b_i, b_j)$ will not be updated further. Thus, when b_j receives a STABLE message from all of its neighbors other than b_i, it sends a STABLE message to b_i.

When b_i receives STABLE messages from *all* its neighbors, it decides that its subgraph eccentricity values with respect to all of its neighbors has stabilized. It can now compute its eccentricity value as the maximum of its subgraph eccentricity values with respect to each of its neighbors. Formally,
$Ecc(b_i) = Max_{(b_x \in NbrList(b_i))}(SEcc(b_i, b_x))$.

Figure 2 illustrates the eccentricity determination phase on a sample routing AG consisting of 6 brokers. It is assumed that all the brokers have received the INIT messages, and the leaf brokers are about to start the process of sending UPDATE messages. All $SEcc$ values at each broker are initialized to zero. For better comprehensibility, the figure shows the three leaf brokers sending out their UPDATE messages *sequentially*. However, it is important to note that the algorithm itself does not place any such sequentiality restrictions. Further, for simplicity, we do not show STABLE messages. Figure 2-A shows the propagation of UPDATE message from b_1. The figure also shows the $SEcc$ values of every broker with respect to all of its neighbors. These are values resulting from the propagation of b_1's UPDATE message. Similarly, Figure 2-B shows the dissemination of an UPDATE message from b_6 and the various $SEcc$ values resulting from this dissemination. Notice that b_6's message is not propagated to b_4, because $SEcc(b_4, b_3)$ is already 3, and it would not be altered by b_6's message. Similarly, in Figure 2-C b_4's UPDATE message is not propagated beyond b_3. Figure 2-C also shows the eccentricity values of all the brokers in the routing AG.

4.1.3 *Center Determination Phase*

Once a node discovers its eccentricity, it sends its eccentricity value to all of its neighbors. A broker whose eccentricity is less than or equal to the eccentricities of all of its neighbors determines that it is one among possibly many centers of the routing AG. Upon discovering that it is a center of the routing AG, the broker sends an announcement message to all its neighbors, which is then propagated through the routing AG. Upon completion of this phase, every broker in the routing AG knows about all the centers of the routing AG. Note that a routing AG might have more than one center. However, if it has multiple centers, these centers form a connected subgraph (i.e., any broker that lies in between two center brokers should itself be a center). The center broker is anointed as the event gatherer for the routing AG. If there are multiple centers, ties may be broken in favor of the broker with the smallest id. Notice that in Figure 2-C, b_3, having the least eccentricity, discovers itself as the center of the routing AG.

4.2 Proof and Analysis of Algorithm

We outline the termination and correctness proofs of our distributed algorithm to find the center of a routing AG. Furthermore, we theoretically analyze its messaging costs.

4.2.1 *Termination*

Recall that each broker sends a single INIT message to its neighbors in the first phase. Similarly, in the center determination phase, brokers send a single message to their neighbors to communicate their respective eccentricities. Further, the brokers also send a single message to their neighbors for disseminating the center information. Thus, we just need to show that the eccentricity determination phase of the algorithm always terminates, which we do by induction.

Recall that the eccentricity determination phase concludes when every broker receives a STABLE message for each of its neighboring brokers. Without loss of generality consider two neighboring brokers b_i and b_j. We will now show that b_i receives a STABLE message from b_j within a finite time. We distinguish between two cases, namely b_j being a leaf broker and b_j being an intermediate

broker. If b_j is a leaf broker (the base case), it sends a STABLE to b_i message as soon as it sends the only UPDATE message to b_i.

In case b_j is not a leaf node, we use induction. The routing AG can be divided into two subgraphs − a subgraph anchored by (b_i, b_j), (denoted $SG(b_i, b_j)$) and a subgraph anchored through (b_j, b_i) (denoted $SG(b_j, b_i)$). Recall that the $SG(b_i, b_j)$ consists of nodes that are reachable from b_i by only passing only through b_j. In our algorithm, b_j sends a STABLE message to b_i as soon as it receives a STABLE message from all of its neighbors other than b_i. Notice that except for b_i, all other neighbors of b_j lie in $SG(b_i, b_j)$. Thus, the act of b_j sending a STABLE message to b_i can recursively depend upon the state of the brokers in $SG(b_i, b_j)$, but it can never depend upon any broker in $SG(b_j, b_i)$. The same argument can now be inductively applied for each neighbor of b_j and the subgraphs anchored by the corresponding edges. Observe that with each successive application of the logic we are dealing with a smaller subgraph. Since the routing AG has finite number of nodes, eventually the subgraphs contain only leaf nodes, which is covered by the base case. Hence, within a finite number of messages, b_j will send a STABLE message to b_i.

4.2.2 Algorithm Correctness

We prove the correctness of our algorithm in two steps. First, we show that our algorithm ensures that every broker in the routing AG correctly discovers its eccentricity. Then, we outline the correctness argument for our center determination protocol (involving eccentricity value exchanges among neighboring nodes). Recall that if b_i and b_j are neighboring brokers $SG(b_j, b_i)$ denotes the subgraph anchored by (b_j, b_i) and $SEcc(b_i, b_j)$ represents the subgraph eccentricity of b_i with respect to b_j. $Ecc(b_i)$ represents the eccentricity of b_i, $NbrsList(b_i)$ denotes the neighbor of b_i in the routing AG, and $Dist(b_i, b_k)$ represents the distance between b_i and b_k.

The correctness argument for eccentricity determination phase itself involves two steps. In the first, we show that in an undirected acyclic graph, the eccentricity of any node can be correctly computed as the maximum of its subgraph eccentricities with respect to all of its neighbors. In the second step we demonstrate that through the proposed algorithm, any arbitrary node in the routing AG can correctly determine its subgraph eccentricity with respect to every neighbor. Consider an arbitrary node b_i in the routing AG. Let b_k be the most distant node from b_i in the routing AG (thus the distance between b_i and b_k is b_i's eccentricity). Suppose b_k lies in $SG(b_i, b_j)$, then $SEcc(b_i, b_j) = Dist(b_i, b_k)$. The acyclic nature of the routing graph implies that there is exactly one path from b_i to b_k in the routing path, and that path entirely lies in the subgraph $SG(b_i, b_j) \cup b_i$. Thus, the distance between b_i and b_k in subgraph $SG(b_i, b_j) \cup b_i$ is the same as the distance between them in the original routing AG. The acyclic nature also implies there is also no other node in $SG(b_i, b_j)$ that is at a greater distance from b_i than b_k. Further, the acyclic property can be used to show that $SEcc(b_i, b_x) \leq Dist(b_i, b_k); \forall b_x \in NbrList(b_i)$. Thus, $Max_{b_x \in NbrList(b_i)}(SEcc(b_i, b_x)) = Dist(b_i, b_k) = Ecc(b_i)$.

Now, we demonstrate that our algorithm ensures that every broker correctly discovers all of its $SEcc$ values. Again, consider two neighboring brokers b_i and b_j. Observe that if b_j forwards an UPDATE message originating at an arbitrary leaf node (say b_l) to b_i, then that value of that message would be $Dist(b_i, b_l)$. Next, if b_k is the most distant node in $SG(b_i, b_j)$[4], then the UPDATE message from b_k would be eventually forwarded by b_j to b_i. The only scenario in which b_j does not forward the message from b_k would be when it has already received a message with a higher value from another node in $SG(b_i, b_j)$. However, this cannot happen since b_k is the most distant node in $SG(b_i, b_j)$. Thus, b_i correctly discovers $SEcc(b_i, b_j)$.

Next, we outline the correctness proof for our center determination protocol. Recall that in the third phase each broker compares its eccentricity information with all of its neighbors, and a node b_i discovers that it is the center if the satisfies the twin conditions ($Ecc(b_i) \leq Ecc(b_x); \forall b_x \in NbrsList(b_i)$ and $\exists b_y \in NbrsList(b_i)$ such that $Ecc(b_i) < Ecc(b_y)$). The correctness proof is based on the following three lemmas, whose proofs we omit due to space limitations. (1) In an acyclic graph, at least one neighbor of a center node has higher eccentricity that the center node. (2) For two arbitrary non-neighbor nodes b_i and b_j of a connected acyclic graph, if b_x is a node in the path between them, then it satisfies the condition $Ecc(b_x) \leq Max(Ecc(b_i), Ecc(b_j))$. (3) In an acyclic graph, if the node b_k is the most distant node to b_i, then the path from b_i to b_k always passes through at least one graph center. Using these three lemmas we can show that for node eccentricities of a connected acyclic graph, the local minima is also the global minimum. This implies that the center nodes and only the center nodes satisfy the two conditions.

4.2.3 Algorithm Analysis

Next, we analyze the message costs of our decentralized center determination algorithm. The three phases of the algorithm are considered separately, and for each phase we try to determine an upper bound for the number of messages circulated in the routing AG.

In the initiation phase, the initiation message sent by one of the nodes is sent to all brokers. If there are M brokers in the routing AG, the total number of messages circulated in this phase is $(M - 1)$. Thus, the message load for this phase is $O(M)$.

Analyzing the message load in the eccentricity determination phase is a bit tricky. Unlike the first phase, we cannot determine the exact number of messages circulated in the routing AG during this phase. In this phase each leaf broker sends out an UPDATE message, which is then circulated to some brokers in the routing AG. The extent to which an UPDATE message is circulated in the routing AG depends upon the subgraph eccentricity values of various brokers when the message reaches them, which in turn depends upon other UPDATE messages that have traversed through that node. Thus, the exact number of UPDATE messages is nondeterministic. Even if every UPDATE message reaches every single node of the routing AG, the total number of messages circulated during this phase would be no more than $L \times (M-1)$, where L represents the number of leaves in the routing AG. Thus, $L \times (M - 1)$ is an upper bound on the message load for the second phase. However, the actual number of messages circulated during this phase is much smaller.

The third phase involves two distinct operations, namely brokers communicating their eccentricity values to their neighbors and disseminating center information to all brokers in the routing AG. The first operation results in $2 \times (M - 1)$ messages and the second operation results in $(M - 1)$ messages. Therefore the total message load for this phase is $3 \times (M - 1)$. Thus, the message load of the entire algorithm is no more than $(L + 4) \times (M - 1)$, but typically is much smaller.

5. AGELE IMPLEMENTATION

This section describes the *Agele* implementation. *Agele* is built on top of the *Siena* simulator [10]. It implements the graph center determination algorithm discussed in Section 4. It also elimi-

[4]For simplicity, we assume that there is only one most distant node in $SG(b_i, b_j)$. However, this assumption can be easily relaxed.

nates redundant and partial messages. For completeness, we first describe briefly the *Siena* system. Then, we describe *Agele*.

5.1 Siena

Siena (Scalable Internet Event Notification Architectures) is a wide-area event-based notification system [10]. It was developed at the University of Colorado. In *Siena*, a message consists of a set of typed attributes. Each attribute has a unique name, type and value. A predicate consists of the disjunction of conjunctions of elementary constraints, where each element constraint is a quadruple (name, type, operator, value). We also adapt the combined broadcast and content-based (CBCB) routing scheme [11] for a content based network. In CBCB routing, the broadcast tree of each message is pruned so that the message is only propagated to those nodes that issued the matching predicates. The routing information in the content based network is propagated in the network using push and pull mechanisms. In the push mechanism, the subscribers push the Receiver Advertisement (RA) to the potential publishers. This sets the routing table along the path, which allows the messages to be forwarded to the subscribers. In the pull mechanism, Sender Requests (SRs) and Update Replies (URs) are applied to maintain the routing table. The router sends the SRs to pull a content based address from other routers, and the routers that are queried reply with URs that contain updated content based address.

5.2 Agele

We added significant infrastructure to *Siena* to create *Agele*. First, we added the center-finding algorithm given in the previous section. This algorithm is run at the start of the simulation, concurrently while the simulation is proceeding. In other words, eliminating redundant and partial messages only occurs after the graph center is found. (However, the center is found quickly, so this effect is minor.)

To implement the *Agele* center-finding algorithm, we construct a connected acyclic overlay for each topic on top of the network of brokers. The algorithm runs on the overlay to find the graph center. All control messages related to the establishment of the graph center are routed through the overlay. Upon the termination of graph center algorithm, each node will have a forwarding table that contains the information to force messages originated from publishers to be routed through the graph center. Once the messages are processed at the graph center, the processed messages are propagated to the subscriber using the content based network routing table [11].

Second, we implemented elimination of redundant and partial messages. To do this, we added a buffer at the center node. When a message M_a arrives at the center, it is buffered. The idea is that a message will be buffered for a period of time, during which incoming messages will be compared for redundancy; however, incoming messages will be considered for merging for no more than (but possibly less than) the buffering time. Two different timers are kept: one is set to T_r, the redundant threshold, and the other to T_m, the merge threshold. The redundant threshold must always be at least as large as the merge threshold (see below).

If a message M_b arrives before the redundant timer expires, and M_b is a subset of M_a, then message M_b is dropped. On the other hand, if M_b is not a subset, and it can be merged with M_a, then a message M_{ab} is created and buffered. In addition, M_a and M_b are removed from the buffer.

On the other hand, if no mergeable message arrives before the merge timer for M_a expires, then M_a is forwarded on from the center node to the next node (according to the forwarding table). However, note that M_a remains in the buffer of the center node. It is removed only when the redundant threshold is reached. As an optimization, if a complete message is ever received at the graph center, it is immediately forwarded, independent of either timer.

6. EXPERIMENTAL RESULTS

This section reports the results obtained using *Agele*. We first describe the experimental setup and then move on to specific results.

6.1 Setup

Our experiments were set up as follows. 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 (time window in which the messages of an event are generated), however, is chosen to be such that it falls within T_b, which we define as the maximum amount of time that any event can take (our experiments investigate different values for T_b). Intuitively, real-world events will likely consist of a burst of messages in a relatively short time period. In our experiments, all nodes subscribe once and for any event, 20% of nodes publish messages pertaining to it. The particular event and associated field names are selected according to a uniform random distribution. We have experimented with two kinds of broker overlay topologies, namely random and power-law networks.

We varied separately the merge threshold, T_m, and the redundant threshold, T_r. Note that ideally, T_r would be set to W (the event time window), but it might not be known a priori, and even if it is, buffer capacities might mandate a smaller value. In our experiments, these ranged between 0 and 10 simulated time units, such that $T_m \leq T_r$. The buffer requirement at the center node was quite modest; it was never more than 100 messages, even on simulations of 3200 nodes.

If T_m and T_r are both 0, then the center node becomes merely a "pass-through", and subscribers will incur additional overhead with no benefit to the overall system. Nevertheless, this is useful from a measurement perspective, as it allows us to isolate the overhead for re-routing messages through the center node.

Generally, we examine the two key metrics: what percentage of the messages were suppressed and how much extra time was added due to using a center node and buffering messages. We use the term *suppressed* because a message can be eliminated (as a duplicate) or merged (with a like event); either way, that message does not emerge from the center node. Note that the minimum number of messages that a subscriber can receive is one. That is, if T_m and T_r are both sufficiently large (generally much larger than the values that we use), all messages will be held at the center node until just one message is forwarded on. For the latter, we measure time per event as the difference between the time that the event is fully received by the subscriber and the time that the first message of that event is published. As the event generation is random, all results presented are the median of several test runs (typically 50).

We start by presenting several results with a random graph and random message origination. All results (except where noted) are relative to *Siena*, where there is no notion of a center node and no buffering of messages. Note that in this section, messages suppressed refers to *originated messages* from a publisher. In addition, T_m and T_r are measured in simulated time units.

6.2 Varying T_m and T_r

First, we investigate the effect of varying T_m (see Figure 3) for five different values of T_b (50, 75, 100, 125 and 150 time units). For

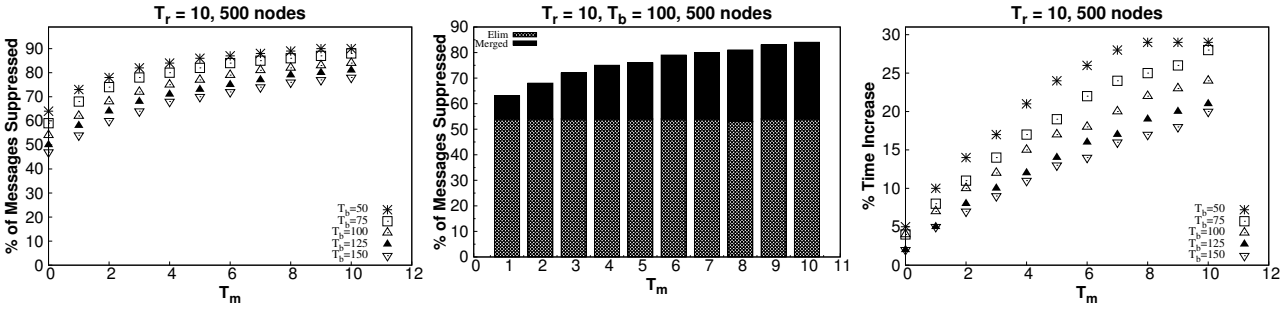
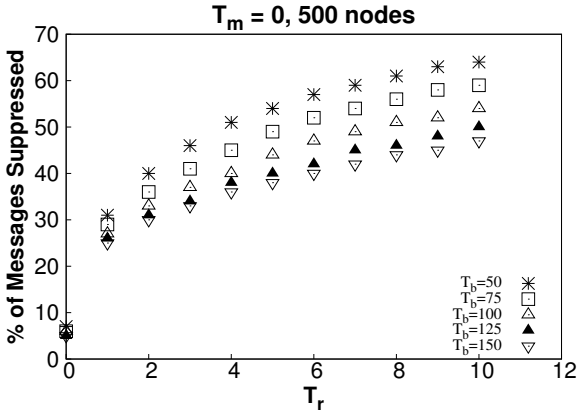


Figure 3: Percentage of messages suppressed, breakdown of whether the suppressed messages were duplicates or merged, and time increase when T_m varies.



T_b	% Time Increase
50	6
75	4
100	4
125	3
150	4

Figure 4: On the left, percentage of messages suppressed when T_r varies. On the right, for $T_r = 10$, the time increase for five different values of T_b.

this experiment, we set T_r to 10 and use 500 nodes. The number of messages suppressed (left-hand graph) is only somewhat dependent on T_m—with the large redundant threshold, many messages are suppressed (especially when T_m is small) even though they conceptually could be merged. (In other words, as one might expect, dropping a message takes priority over merging one.) In addition, the percentage of messages that are suppressed is largest when T_b is smallest. This is expected, as when messages are clustered, there is a greater chance that when one message from a given event arrives at the graph center, other messages from the same event arrive before T_m time units. Note that the percentage of suppressed messages levels off in our range of T_m values because events with few messages cannot be suppressed in practice.

The right-hand graph shows the cost of suppressing messages, which ranges from a 5% to 34% overhead. When T_b is 100, the overhead is quite good—at most 24%. Furthermore, choosing T_m to be 4, we limit the overhead to 15% yet suppress over 70% of the messages. This would likely be a good tradeoff. In a real-world system, one benefit of suppressing—which we are not considering here, but will in future work—is that bandwidth consumption is lower. This in turn will decrease potential message drops due to overloaded nodes. Therefore, the time for a subscriber to receive an event will likely be *lower* when using Agele than baseline Siena.

Second, we investigate the effect of varying T_r (see Figure 4). For this experiment, we set T_m to 0. It is clear that the number of messages suppressed (left-hand graph) is strongly dependent on T_r. The effect is similar as the number of nodes increases. The time increase is independent of T_r, because time overhead occurs only when messages are buffered at the center node for potential merging. Therefore, we show the effect on time increase when varying T_b. The table on the right-hand side of the figure shows that the time increase, for a fixed value of T_r (10 in this case), is only slightly dependent on T_b (which is varied between 50 and 150).

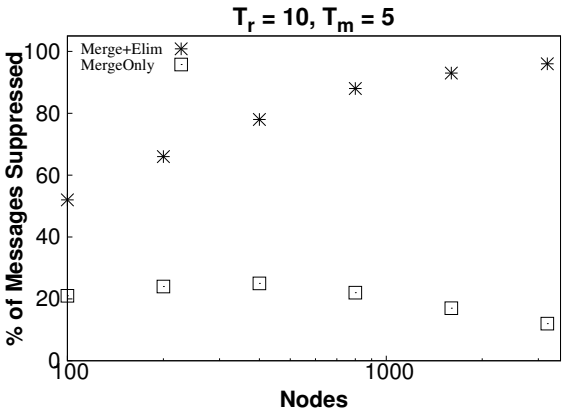
6.3 Scalability

We next investigate the scalability of *Agele* by varying the number of nodes from 100 to 3200, doubling the number each time (see Figure 5). For this experiment, we set T_m to 5 and T_r to 10. The left-hand graph contains two plots. One is the percentage of total messages suppressed, and one is only the number of messages merged. The total messages suppressed *increases* with the number of nodes. This is because there is a fixed number of events, and hence more duplicate messages are suppressed. Note that in a real-world system, increasing the number of human participants will likely only slightly, if at all, increase the number of events, assuming that the geographical area is fixed. The number of messages merged—which is more difficult to scale—remains relatively constant as the number of nodes increases. Not until 1600 nodes does this number start to fall off, and it does so slowly.

The right-hand table shows the time increase relative to Siena. The overhead is between 17% and 32%, which is quite good. Overall, *Agele* scales well especially considering that typical pub-sub systems contain less than 1000 broker nodes.

6.4 Graph Centers

The *Agele* system is predicated on finding the graph center, because this is the most effective node through which messages should be routed. To show the effectiveness of finding the center, we performed the following experiment. We started with a 300-node ran-



Number of Nodes	% Time Increase
100	17
200	32
400	27
800	29
1600	20
3200	22

Figure 5: Percentage of messages suppressed and time increase when the number of nodes varies. The graph uses a log scale for the x-axis.

Eccentricity	Normalized Event Time (s)
7	18
8	53
9	340
10	1263
11	1572

Nodes	Time to Find Graph Center (s)
100	10
200	11
400	15
800	16
1600	19
3200	20

Figure 6: On the left, event times for aggregator nodes using different eccentricities. On the right, the time to find the graph center for various node counts, normalized based on the number of aggregator nodes used.

dom graph, where the center node has an eccentricity of 7. We tried using as aggregators all nodes with eccentricity value 8. We repeated this (separately) using values 9, 10, and 11. The idea is that using centers with larger eccentricities will yield inferior results.

The left-hand side of Figure 6 shows the results. We measured the average time for an event in *Agele* over aggregators with five different eccentricities. Because there are more nodes with eccentricity 8 than 7, 9 than 8, etc., we normalize the results based on the number of centers. (This is because with more aggregators in general means lower event times.) The figure clearly shows that using aggregators with larger eccentricities produces inferior results. Hence, using center nodes to aggregate is critical.

Finally, the right-hand side of Figure 6 presents the time to find the graph center for different numbers of nodes (ranging from 100-3200). This shows clearly that the center finding algorithm within *Agele* is efficient.

7. RELATED WORK

Pub-sub systems have continued to be an important research area over several years [3, 8, 9, 10, 15, 17, 19, 20, 23, 26, 36, 30, 12, 27, 21]. On this basis of how subscriptions are specified, pub-sub systems are classified into topic-based [3, 6], content-based [8, 10, 30, 33], type-based [16], and type- and attribute-based [27] categories. From an architectural standpoint, distributed pub-sub systems [7, 8, 10, 14, 20, 36, 12] provide significantly better scalability than their centralized counterparts [25]. As mentioned previously, the subscription mechanism of *Agele* is similar to type and attribute-based subscription model [27], and it adopts a distributed broker framework.

Few pub-sub systems have even considered the issue of noisy event streams. To the best of our knowledge, none has looked into challenges posed by incomplete (partial) event messages. A few touch upon the duplicate elimination problem [21, 18, 32]. However, these systems are only capable of eliminating *exact* duplicates, and they offer simplistic solutions. Huang and Garcia-Molina [21] relegate this responsibility to the subscriber, an approach which has severe drawbacks. The in-network duplicate elimination scheme used by XTreeNet system [18] has two major limitations. First, it requires each node in the tree to maintain a cache of messages that has recently passed through it. Because nodes often participate in multiple trees, they need to store large number of messages for this scheme to be effective. Second, the technique is not effective in reducing message traffic due to duplicates originating from different regions of the overlay. Our definition of redundancy is broader in the sense that an event message is considered to be duplicate if the information it carries has already been obtained by aggregation of other messages, even though it was not an exact match to any of the previous messages. Thus, our duplication elimination is more powerful. By designating specific event gatherers for every routing AG, we eliminate the need for message caching at each node in the overlay. Furthermore, our technique is effective even when the messages originate in different regions of the network.

Although, event aggregation in decentralized pub-sub systems has similarities to distributed stream processing [5, 22, 24, 28, 29], there are several crucial differences between the two. First, the sources in stream processing systems are generally known when the query plan is evolved and the operators are placed, whereas in a pub-sub system, any publisher that has issued an advertisement can generate a corresponding event. Second, the source nodes in a stream processing environment are assumed to continuously produce data for long durations. In pub-sub system, however, publishers generate events in a non-continuous manner and at arbitrary points in time. Thus, the heavy-weight optimization-based query

planning and operator placement strategies used by stream processing applications are not appropriate for *Agele*. Furthermore, most operator placement strategies require a global view of the network topology, which is not available in our decentralized broker architecture. Complex event detection [5] also bears similarities to event aggregation. However, most of the current approaches to complex event detection rely upon apriori planning which assumes that the event sources are known before hand. Adaikkalavan and Chakravarthy [4] discuss modeling and specification of incomplete events.

In the context of multicast routing, Thaler and Ravishankar [34] propose heuristic-based strategy for finding the graph center, which works in multiple rounds. There are two main differences between their algorithm and ours. First, being a heuristic-based approach, their algorithm may not always locate the exact center. Second, and more importantly, although their scheme does not require a global view of the overlay topology, it assumes that in each round the center knows about all nodes in the multicast group. By contrast, our algorithm does not require centralized membership information, and it always discovers the exact center of the routing AG. The scheme by Song [31] requires each node to first discover the identity of all other nodes, and then execute the all pairs shortest path algorithm. Unfortunately, this straightforward distribution strategy imposes significant computation and communication overheads at all nodes in the network, thus making it impractical for our application.

8. CONCLUSION AND FUTURE WORK

Real world pub-sub systems, especially ones with human publishers, are potentially faced with event data that is fraught with various kinds of noise. Dealing with this noise effectively is critical, yet challenging both from data management and distributed computing perspectives. This paper presents our initial approach towards designing an efficient distributed broker overlay-based pub-sub system that eliminates redundant event messages as well as aggregates information from multiple messages corresponding to the same event.

We introduced the concept of *event gatherer* as a designated broker of a particular routing graph that is responsible for eliminating redundant messages and merging messages containing partial event information through that routing graph. We showed that in order to achieve high efficiency and low overheads, the event gatherer should be located at the graph center of the corresponding routing graph. A novel, completely decentralized algorithm has been presented for discovering the center of an acyclic broker network. The above ideas are incorporated into our system, *Agele*. We have performed several experiments for studying the performance of the *Agele* system under various conditions. Early results show that the proposed techniques are effective and efficient, thereby demonstrating the viability of our approach.

As a part of our ongoing work, we have identified several avenues for further enhancing the scalability and flexibility of *Agele*. First and foremost, in the current design, an event gatherer handles the information aggregation and redundancy elimination responsibilities of all the events of a given topic. The event gatherers can potentially become *hot-spots* in large broker overlays. We plan to address this issue with a two-pronged strategy. One part of this is to, rather than using the center of the routing acyclic graph as the event gatherer of all events of the corresponding topic, identify a set of brokers around the graph center. Any of these brokers can assume the responsibilities of information aggregation and redundancy elimination pertaining to an event of the corresponding topic. The other part is to, instead of individually routing published messages of an event to the corresponding event gatherer, perform partial message aggregation and redundancy elimination along the route from publishers to the event gatherer.

Second, we want to enhance subscriber flexibility in choosing the duration of the aggregation cycle. Currently, the duration of the aggregation cycle (T_m) is a configuration parameter, and all the recipients of an event encounter the same latency in receiving the event. However, in a practical pub-sub system, some subscribers might prefer to receive aggregated events at shorter durations at the expense of higher message loads whereas others might prefer less aggressive notification. Our initial thoughts are to propagate the subscriber-specified aggregation cycle duration values upstream in the routing acyclic graph, while combining them at various intermediate brokers so that the event gatherer adopts the most aggressive aggregation cycle. The aggregated event messages issued by the event gatherer will be held at various brokers and the subscribers will be notified as per their preferred schedule.

Third, we plan to develop efficient techniques to handle changes of the routing structures caused by dynamics of the broker overlay or modifications to the subscriptions. Realizing the above enhancements in a decentralized broker overlay needs new protocols, algorithms and systems-level techniques, the details of which are being currently addressed.

9. ACKNOWLEDGMENTS

This work is partially supported by National Science Foundation under Grant No. 0716357.

Any opinions or findings expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

10. REFERENCES

[1] Digg (http://digg.com).

[2] Twitter (http://twitter.com).

[3] TIB/Rendezvous. White paper, 1999.

[4] Raman Adaikkalavan and Sharma Chakravarthy. Events must be complete in event processing! In *Proceedings of ACM-SAC*, 2008.

[5] Mert Akdere, Ugur Çetintemel, and Nesime Tatbul. Plan-based complex event detection across distributed sources. In *Proceedings of VLDB*, 2008.

[6] Roberto Baldoni, Roberto Beraldi, Vivien Quéma, Leonardo Querzoni, and Sara Tucci Piergiovanni. TERA: topic-based event routing for peer-to-peer architectures. In *Proceedings of DEBS*, 2007.

[7] Roberto Baldoni, Carlo Marchetti, Antonio Virgillito, and Roman Vitenberg. Content-based Publish-Subscribe over Structured Overlay Networks. In *Proceedings ICDCS*, 2005.

[8] 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 *Proceedings of ICDCS 1999*, 1999.

[9] Martin Bauer and Kurt Rothermel. How to Observe Real-World Events through a Distributed World Model. In *Proceedings of ICPADS*, 2004.

[10] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, 2001.

[11] Antonio Carzaniga, Matthew. J. Rutherford, and Alexander L. Wolf. A Routing Scheme for Content-Based Networking. In *Proceedings of INFOCOM 2004*, 2004.

[12] 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.

[13] Jianxia Chen, Lakshmish Ramaswamy, and David K. Lowenthal. Agele: Dealing with redundant and partial events in a real-world publish-subscribe system. Technical Report UGA-CS-TR-09.001, 2009.

[14] Gregory Chockler, Roie Melamed, Yoav Tock, and Roman Vitenberg. Constructing scalable overlays for pub-sub with many topics. In *Proceedings of PODC*, 2007.

[15] Paolo Costa, Matteo Migliavacca, Gian Pietro Picco, and Gianpaolo Cugola. Epidemic Algorithms for Reliable Content-Based Publish-Subscribe: An Evaluation. In *Proceedings of ICDCS*, 2004.

[16] Patrick Th. Eugster, Rachid Guerraoui, and Christian Heide Damm. On Objects and Events. In *Proceedings of OOPSLA*, 2001.

[17] Patrick Th Pascal Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The Many Faces of Publish/Subscribe. *ACM Computing Surveys*, 35(2), 2003.

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

[19] Ludger Fiege, Mariano Cilia, Gero Mühl, and Alejandro P. Buchmann. Publish-Subscribe Grows Up: Support for Management, Visibility Control, and Heterogeneity. *IEEE Internet Computing*, 10(1), 2006.

[20] Abhishek Gupta, Ozgur D. Sahin, Divyakant Agrawal, and Amr El Abbadi. Meghdoot: content-based publish/subscribe over P2P networks. In *Middleware 2004*, 2004.

[21] Yongqiang Huang and Hector Garcia-Molina. Publish/subscribe in a mobile environment. *Wireless Networks*, 10(6), 2004.

[22] Navendu Jain, Michael Dahlin, Yin Zhang, Dmitry Kit, Prince Mahajan, and Praveen Yalagandula. STAR: Self-Tuning Aggregation for Scalable Monitoring. In *Proceedings of VLDB*, 2007.

[23] Zbigniew Jerzak and Christof Fetzer. Bloom Filter Based Routing for Content-based Publish/Subscribe. In *Proceedings of DEBS*, 2008.

[24] Oana Jurca, Sebastian Michel, Alexandre Herrmann, and Karl Aberer. Query Driven Operator Placement for Complex Event Detection over Data Streams. In *Proceedings of EuroSSC*, 2008.

[25] R. Lewis. Advanced Messaging with MSMQ and MQSeries, 1999.

[26] José Mocito, J. Alfonso Briones-García, Boris Koldehofe, Hugo Miranda, and Luís Rodrigues. Geographical Distribution of Subscriptions for Content-based Publish/Subscribe in MANETs. In *Middleware (Companion)*, 2008.

[27] Peter Pietzuch and Jean Bacon. Hermes: A Distributed Event-Based Middleware Architecture. In *Proceedings DEBS*, 2002.

[28] Peter R. Pietzuch, Jonathan Ledlie, Jeffrey Shneidman, Mema Roussopoulos, Matt Welsh, and Margo I. Seltzer. Network-Aware Operator Placement for Stream-Processing Systems. In *Proceedings of ICDE*, 2006.

[29] Thomas Repantis, Xiaohui Gu, and Vana Kalogeraki. Synergy: Sharing-Aware Component Composition for Distributed Stream Processing Systems. In *Proceedgins of Middleware*, 2006.

[30] Bill Segall, David Arnold, Julian Boot, Michael Henderson, and Ted Phelps. Content Based Routing with Elvin4. In *Proceedings of AUUG2k*, 2000.

[31] Linlin Song. A Distributed Algorithm for Graph Center Problem. Master's thesis, 2003.

[32] Mudhakar Srivatsa and Ling Liu. Securing Publish-Subscribe Overlay Services With EventGuard. In *Proceedings of ACM-CCS*, 2005.

[33] Sasu Tarkoma. Dynamic content-based channels: meeting in the middle. In *Proceedings of DEBS*, 2008.

[34] David Thaler and Chinya V. Ravishankar. Distributed Center-Location Algorithms. *IEEE Journal on Selected Areas in Communications*, 15(3), 1997.

[35] Robert Voigt, Robert Barton, and Shridhar Shukla. A Tool for Configuring Multicast Data Distribution Over Global Networks. In *Proceedings of INET*, 1995.

[36] Spyros Voulgaris, Etienne Riviere, Anne-Marie Kermarrec, and Maarten van Steen. Sub-2-Sub: Self-Organizing Content-Based Publish Subscribe for Dynamic Large Scale Collaborative Networks. In *Proceedings of the 5th international workshop on peer-to-peer systems*, Feb 2006.

[37] David Wall. *Mechanisms for Broadcast and Selective Broadcast*. PhD thesis, Stanford University, 1980.