Distributed Event Aggregation for Publish/Subscribe Systems

Technical Report

Navneet Kumar Pandey*, Kaiwen Zhang†, Stéphane Weiss*, Hans-Arno Jacobsen†, Roman Vitenberg*
*Department of Informatics, University of Oslo
†Department of Computer Science, University of Toronto
*{navneet,stephawe,roman}@ifi.uio.no,†{kzhang,arno}@msrg.utoronto.ca

***Abstract*—Modern data-intensive applications handling massive event streams such as real-time traffic monitoring require support for both rich data filtering and aggregation capabilities. While the pub/sub communication paradigm provides an effective solution for the sought semantic diversity of event filtering, the event processing capabilities of existing pub/sub systems are restricted to singular event matching without support for stream aggregation, which can be accomodated only via the end-to-end principle.**

In this paper, we propose the first systematic solution for supporting a range of time-based aggregation semantics in a pub/sub system. In order to eschew the need to disseminate a large number of publications to the subscriber, we strive to distribute the aggregation computation within the pub/sub overlay network. By enriching the pub/sub language with aggregation semantics, we allow pub/sub brokers to aggregate incoming publications without forwarding them to the next broker downstream. We show that our baseline solutions, one which aggregates early (at the publisher edge) and another which aggregates late (at the subscriber edge), are not optimal strategies for minimizing bandwidth consumption. We thus propose an adaptive rate-based heuristic solution which determines which brokers should aggregate publications. We show that this adaptive solution leads to improved performance compared to our baseline solutions using real datasets extracted from our traffic monitoring use case.

I. Introduction

Provision for aggregating information is a key element of many distributed systems and applications nowadays [1], [2], [3], [4]. In this paper, we introduce mechanisms for supporting aggregate subscription in the context of the popular pub/sub messaging paradigm [5], [6]. Pub/Sub has been widely used in business process execution [7], workflow management [8], business activity monitoring [9], stock-market monitoring [10], selective information dissemination and RSS filtering [11], complex event processing for algorithmic trading [12], social interaction [13] and network monitoring and management [9].

In order to achieve scalability and high throughput, pub/sub has traditionally focused on performance over extended functionality. In particular, aggregation computations are not supported by typical pub/sub systems, in contrast to other communication paradigms, such as stream processing. At the same time, providing support for aggregation in pub/sub is both useful and it poses a unique set of challenges, as we elaborate on below.

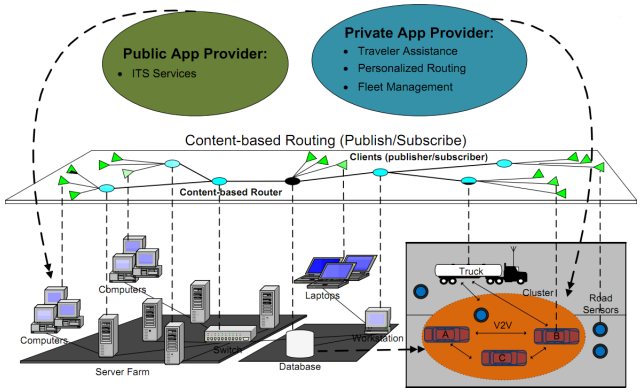


Figure 1. Conceptual CVST architecture

Often, data-intensive applications for pub/sub may benefit from aggregation over a filtered stream [14], [15], [16], [17], [18]. As a specific illustrating scenario, we consider an intelligent transport system (ITS) [19] (see Fig. 1) within the framework of the Connected Vehicle and Smart Transportation (CVST) project [20]. This system disseminates road sensor data to ITS server farms to coordinate traffic flow and invoke appropriate agents (e.g., police, first responders, radio networks, smartphones etc.) on demand. Besides, smart vehicles are connected to each other (Vehicle-to-Vehicle) to assist the drivers and mitigate the chance of collision [21].

This application requires a broad variety of filtering capabilities in order to allow each data sink to express its interests. Furthermore, the heterogeneous and mobile nature of the agents calls for a loosely coupled communication paradigm. These properties indicate that a pub/sub (specifically content-based) communication substrate would be a good fit for this application [6].

At the same time, this application implies a lot of queries related to real-time monitoring that require aggregation over time-based windows. For instance, road sensors output the speed of every passing vehicle along a certain road segment during a certain time period [1]. These data points should be aggregated to compute the average speed in that segment.

Supporting aggregation in pub/sub, however, leads to a new set of challenges. The solution should be compatible with and pluggable into existing pub/sub systems. Unlike most existing substrates for distributed aggregation, it should support a large scale of data distribution and dynamic loosely-coupled matching between data sinks and sources.

It should be integrated with existing pub/sub infrastructures, such as hierarchies of message brokers, which impose their own logic for routing and flow control.

Since the same publication data may contribute to many aggregate subscriptions, which are spread over dispersed data sinks, an important aspect of aggregation in pub/sub is optimizing data flows from multiple sources to multiple sinks across data brokers. This emphasis is different from data aggregation in other types of distributed systems, where the focus is on an efficient query processing engine for complex event patterns which scales well in terms of the number of events and queries. The challenge of optimizing data flows is exacerbated by the fact that the same data can be used for aggregate and regular subscriptions. Finally, irregular event rates at the publishers, coupled with dynamic content-based matching and injection of subscriptions on the fly make it difficult to predict data flows in advance. This necessitates adaptive solutions with respect to aggregating and forwarding the information.

In this paper, we show the need for adaptation in pub/sub aggregation by considering two extreme baselines. First, we propose to aggregate at the subscriber edge broker. This solution is the straightforward way to support aggregation where publications of interest are collected at the subscriber's endpoint, which requires the pub/sub system to deliver the entire stream of matching publications. Second, we consider an alternative approach that aggregates data at the publisher edge brokers: any broker directly attached to a publisher computes partial results over all matching publications received, while brokers downstream merge partial results coming from different upstream brokers. We show that, depending on pub/sub parameters which are changing over time (e.g. number of subscriptions, publication matching rate, etc), one approach outperforms the other.

As our solution, we propose an adaptive algorithm, which measures the rate of publications and notifications and switches between publication forwarding and aggregation computation according to a cost model. The protocol we propose is lightweight in nature as it does not require any changes to the system model of current pub/sub implementations. Any additional information to be exchanged by brokers is piggybacked onto existing traffic.

In summary, the contributions of this paper are as follows:

- We provide two baseline algorithms to support aggregation using pub/sub. The first algorithm employs subscriber-edge aggregation (late approach), where the entire matching publication stream is disseminated to the subscriber edge brokers where the results are computed. The second baseline adopts the early aggregation approach by performing aggregation at the publisher edge brokers and forwarding partial results downstream.
- We evaluate the performance of our adaptive solution and the baselines using real traces collected from our traffic monitoring use case along with a dataset from a stock market application. We show that the performance of both baselines vis-à-vis each other depends on the properties of the workload, while the adaptive solution outperforms both in every situation.

II. Aggregation and pub/sub background

In this section, we describe the content-based pub/sub model after providing a taxonomy of aggregation semantics relevant for this model.

A. Aggregation taxonomy

Aggregation refers to the summarization of multiple data points into a compact representation [4]. An aggregation function $f : \mathbb{N}^I \rightarrow O$ takes a multiset of elements of input type I and produces a results of output type O. Example aggregation functions include `Average`, `Minimum`, `Sum` and `Top-k`.

Decomposability: An aggregation function is considered decomposable if the same result can be obtained by aggregating the partial results over subsets of the original input. Essentially, decomposability allows the aggregation computation to be distributed by partitioning the input set. Amongst decomposable functions, there is a distinction between *self-decomposable* and *indirectly decomposable* functions. The self-decomposable property is defined in [4] as follows:

Definition 1. ***(Self-Decomposable Aggregation Function)*** *An aggregation function $f : \mathbb{N}^I \rightarrow O$ is called self-decomposable if for all non-empty multisets X and Y there exists an operator $\diamond$ such as :*

$$f(X \uplus Y) = f(X) \diamond f(Y)$$

where $\uplus$ is the standard multiset sum.

For example, `Count` is self-decomposable since `Count`$(X \uplus Y)$ = `Sum`(`Count`(X), `Count`(Y)). In this case, `Sum` is used as the merge operator $\diamond$. *Indirect decomposable* functions are not self-decomposable but they can be transformed into an expression containing only self decomposable functions. For instance `Average` is indirectly decomposable: it is not self-decomposable but it can be expressed as the `Sum` divided by the `Count`.

Non-decomposable functions cannot be broken down: the result can only be derived by taking the entire input set and processing it at once. For instance, `Distinct Count` (i.e., cardinality of a set) is non-decomposable: we require the entire set of elements to avoid counting duplicates.

Duplicate sensitivity: A duplicate sensitive aggregation operation is affected by duplicate elements while an insensitive operation is not. For instance, `Maximum` is a duplicate insensitive operation while `Count` is not.

To support duplicate sensitive operators, we need to ensure that each element is aggregated exactly once. This is a challenge for distributed environments where the aggregation of an element can occur at different locations.

B. Window parameters

Given a stream of values, aggregation functions are computed repeatedly over a sequence of windows, each with a start point and duration. These window properties can be defined using time (*time-based* windows) or number

of elements (*count-based* windows) [4]. The start point is determined by the window shift size and the start point of the previous window. The window shift size (δ) and duration (ω) are expressed either in terms of time or number of values. If $\delta < \omega$, consecutive window ranges will be overlapping (*sliding*). If $\delta = \omega$, the window is called *tumbling*, otherwise ($\delta > \omega$), it is called *sampling*.

C. Publish/Subscribe model

Our work focuses on content-based pub/sub using advertisement-based routing [5]. In this model, each publisher and subscriber is attached to a single broker. Brokers are in charge of forwarding publications from publishers to the appropriate subscribers. Routing paths are initialized through the use of advertisements, which are flooded through the network. Subscriptions are matched against those advertisements to build a delivery path from the originating publisher to the subscribers. Publications are then matched against the subscriptions and forwarded down the appropriate nodes.

Content-based matching allows for predicate-based filtering on a per-publication basis. In other words, matching is stateless: each publication is matched in isolation against the subscriptions and is not affected by the existence of other publications.

III. Distributed aggregation for pub/sub

In this section, we discuss the scope of our solution before describing the steps involved in the aggregation process. These steps can be distributed among several brokers, and we present two possible approaches called "late aggregation" and "early aggregation". Finally, we argue that there is no clear winner between late and early aggregation, and thus we propose an adaptive solution that distributes the aggregation steps among brokers to limit the number of messages exchanged.

A. Scope of the solution

Our aggregation solution supports all classes of decomposability and duplicate sensitivity properties. In terms of notification frequency, we support all three (sliding, tumbling and sampling) window types. Moreover, the system can process workloads containing a mix of both regular (non-aggregate) and aggregate subscriptions. However, we limit the scope of this paper to *time-based* semantics and consider only acyclic pub/sub overlay networks. Our primary focus for this work is to reduce inter-broker communication traffic. We also show that reducing traffic reduces the load on the brokers. On the other hand, we do not focus on optimizing the subscription overlaps based on their predicates or aggregation semantics: multi-query optimization techniques from the distributed stream management systems field can be applied here [22].

We extend the pub/sub subscription semantics with aggregate subscriptions. In addition to the standard conjunction of predicates, the subscribers need to specify the operator, the attribute over which aggregation is performed, and the window parameters (ω, δ). For example, a subscription to the moving average over one hour at every 10 minutes for the stock symbol '*IBM*' would be expressed as {sym='*IBM*', op='`Average`', par='*value*', ω ='*1h*', δ ='*10 min*'}.

B. Aggregation flow

This section presents the overall process for aggregation which is general and applicable to all three type of operators (i.e. self-decomposable, indirect decomposable and non-decomposable). To illustrate the aggregation process, let consider the following three subscriptions S_1, S_2, and S_3, each having a different operator type:

- S_1={sym='*IBM*', op='`Sum`', par='*value*', ω='*1h*', δ='*10 min*'}
- S_2={sym='*IBM*', op='`Average`', par='*value*', ω='*1h*', δ='*10 min*'}
- S_3={sym='*IBM*', op='`Median`', par='*value*', ω='*1h*', δ='*10 min*'}

For the examples of this section, we assume that five publications are generated in a 10 minutes window for the symbol '*IBM*' with a '*value*' attribute of 4, 2, 5, 9 and 5.

The overall process is divided into the steps given below.

(1) **Subscription transformation:** For operators which are not self-decomposable, the subscription needs to be transformed before being propagated. For example, the non-decomposable subscription S_3 is transformed to S'_3: {sym='*IBM*', op='`Fetch`', par='*value*', ω='*1h*', δ='*10 min*'} where "`Fetch`" is an auxiliary operator that batches all matching publications. A second example is the indirect decomposable subscription S_2 (with the `Average` operator) which is transformed into a subscription containing self-decomposable operators (`Sum` and `Count`). S_2 becomes S'_2: {sym='*IBM*', op='`Sum`', op='`Count`', par='*value*', ω='*1h*', δ='*10 min*'}.

(2) **Subscription dissemination:** Disseminating aggregate subscriptions follows the same steps as regular one.

(3) **Aggregation of publications and collection of results:** In this step, for incoming matching publications, the broker computes results on a per-window basis. Depending upon the technique used, some broker will collect or merge the values obtained via incoming partial result streams. For example, the result computed for S_1 is 25, for S'_2 is ($\texttt{sum} = 25, \texttt{count} = 5$), and for S'_3 we obtain $\{4, 2, 5, 9, 5\}$.

(4) **Transformation of results and notification:** For non-decomposable and indirect decomposable operations, results obtained from the previous step must be transformed back to fit the subscription issued by the client. Hence, for S_2, the average of 5 is computed as ($\texttt{sum} = 25, \texttt{count} = 5$) out of S'_2. Similarly for S_3, the result 5 is computed from publications obtained for S'_3. The final step consist of sending the results to the subscribers, 25 for S_1, 5 for S_2 and 5 for S_3.

C. Publication process for aggregation

This section describes the steps of the aggregation procedure performed at a broker upon receiving a publication.

(1) **Subscription matching:** Upon receiving a publication, the broker matches it against aggregate subscriptions

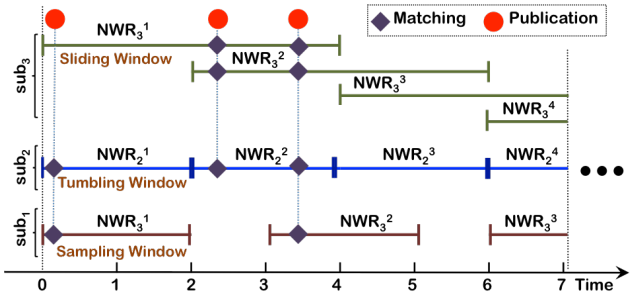


Figure 2. Notification Window Ranges (NWRs)

based on the conditional predicates. This matching is identical to the one performed for regular subscriptions using the pub/sub matching engine.

(2) **Finding Notification Window Ranges (NWRs):** From a matched aggregate subscription, depending upon its window and shift size, zero or more notifications are generated (see Fig. 2). We define an instance of Notification Window Range (NWR) as a triplet of $\langle$*subscription*, *start-time*, *duration*$\rangle$. NWRs are used to manage notifications. Fig. 2 illustrates the concept of NWR. In this example, we consider three aggregate subscriptions sub_1, sub_2 and sub_3 which are respectively for sliding, tumbling and sampling window. The first publication (around time 0) matches the subscriptions sub_1 and sub_2 only. The second publication matches sub_2 and sub_3. As sub_3 has a sliding window, this publication matches two NWRs for sub_3 (NWR_3^1 and NWR_3^2) and one NWR for sub_2 (NWR_2^2). Finally, the last publication matches all three subscriptions and contributes to all four NWRs.

(3) **Computing:** Once an NWR ends, the result is computed according to the aggregation operator.

(4) **Propagating the results:** Computed results are propagated towards the appropriate subscribers using the regular routing mechanisms employed by the pub/sub system.

Note that publications are timestamped by their publisher edge broker. These timestamps determine their corresponding unique NWRs. Consequently, an identical aggregate subscription issued by two different subscribers receive the same notifications.

D. Late aggregation

Late aggregation can be considered as a naive approach for aggregation in pub/sub systems. In acyclic overlays, each subscriber is connected to a pub/sub system via a subscriber edge broker. Therefore, disseminating all matching publications to this sink broker and aggregating them there produces a correct result stream, since each matching publication is received and processed exactly once by the sink broker. This forms the basis for our late aggregation solution.

To illustrate the behavior of the naive approach, consider the example described in Fig. 3(a). A subscriber attached to broker B_4 sends a subscription requesting the average of the value of a stock symbol IBM: {sym='*IBM*', op='`Average`', par='*value*', ω='*1h*', δ='*10 min*'}. As explained in Sect. III-B, this subscription is transformed into {sym='*IBM*', op='`Count`', op='`Sum`', par='*value*', ω='*1h*', δ='*10 min*'} and then propagated towards the publishers. As shown in Fig. 3(a), publisher at broker B_1 generates two publications P_1 and P_2 having values '5' and '9' correspondingly. For the sake of simplicity, we assume that these publication matched to single NWR. Similarly, the second publisher at B_2 also generates publications P_3 and P_4 having respectively '4' and '2' as values for the same NWR. These publications are routed based on their content until they reach broker B_4. Upon reception of P_1, P_2, P_3 and P_4, B_4 computes the sum '20' and the count '4' used to generate the result notification containing the average '5' for the subscriber.

Performing aggregation at the subscriber edge broker does not require any modification to the core pub/sub logic. We follow the end-to-end principle and install aggregation components at the edges while the rest of the brokers are unaware of aggregation semantics.

E. Early aggregation

Early aggregation approach consists of aggregating publications as early as possible, i.e., at the brokers closest to the publishers. Unlike late aggregation, the computation is distributed among a set of brokers and thus requires collection of intermediate results as presented below. We refer to the results computed at individual brokers as *Intermediate Results (IR)* while crafted publications carrying these results are called *IR publications*.

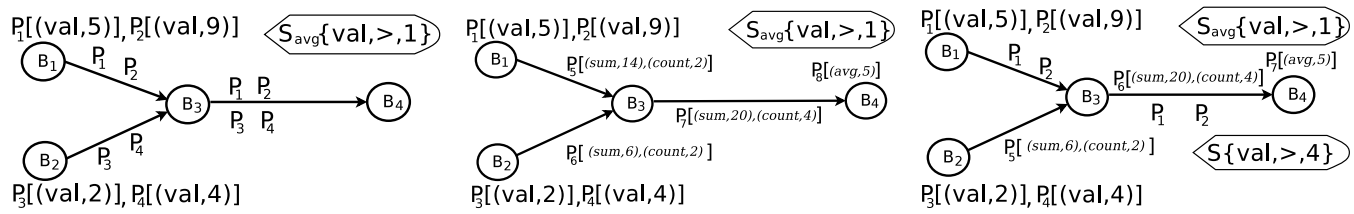
Intermediate brokers may expects IR publication from multiple source brokers concurrently. Processing of IR publications differs from regular processing as IR publications contain partial results, window information etc. For these, an intermediate broker follows a separate processing sequence. To differentiate it from the publication process presented in Sect. III-C, we refer to the former as Initial Publication processing Sequence (IPS) (as it applies only once at the beginning of notification route) and to the latter as Recurrent Publication processing Sequence (RPS) (applies repeatedly at each broker in the notification route).

The RPS consists of the following steps:

(1) **Finding the matching NWR:** Upon receipt of an IR publication, a broker finds the matching NWR for it.

(2) **Buffering the intermediate results:** IR publications matching the same NWR may not arrive simultaneously due to the asynchronous nature of pub/sub communication. Therefore, IR publication requires buffering while waiting for additional results. The initial insertion of an IR publication sets a countdown timer for the corresponding NWR buffer.

(3) **Synchronizing and merging IRs for the same NWR:** After expiration of a timeout, collected IRs matching the same NWR are merged using an operator which can differ from the actual aggregation operation. For instance, two streams of intermediate `Count` results can be combined together by summing the counts of the matching windows.



(a) Late aggregation (Naive approach) (b) Early aggregation (c) Adaptive Aggregation

Figure 3. Aggregation techniques

(4) **Propagating merged intermediate results:** A merged IR is then routed to the downstream brokers. If the next hop is again an intermediary broker, RPS is activated again at that broker to collect and merge more intermediary results. Therefore, a stream of IR publications goes recurrently through RPS. At the end, the result is converted to its final form before being delivered to the client.

Similarly to the late aggregation algorithm, subscription transformation happens at the subscriber edge broker. However, the publication process sequence described in Sect. III-C is now executed at the source brokers leading to the generation of several IRs for an NWR. Every downstream broker that expects to receive aggregate publications from more than one source, performs the RPS.

Figure 3(b) shows how early aggregation works on a simple scenario. The publications P_1 and P_2 are processed by the broker B_1 which results in an IR publication P_5 with the intermediate result: [(`sum`=14, `count`=2)]. Similarly on broker B_2, an IR publication P_6: [(`sum`=6, `count`=2)] is produced. When the broker B_3 receives P_5 and P_6, the RPS is applied and produces the result P_7: [(`sum`=20, `count`=4)]. Upon receipt of this intermediary result, the broker B_4 produces a notification for the subscriber. In this scenario, the overall number of messages is reduced from 8 to 3 when comparing early to late aggregation.

F. Motivation for adaptation

Early aggregation greedily reduces the number of messages exchanged among brokers by performing the aggregation as close to the publishers as possible. However, our experiments (see Fig. 4(a)) show that early aggregation generates more messages than the late approach in some scenarios. The comparative performance of early vs late aggregation depends upon several factors such as:

window shift size: In some scenarios such as sliding NWRs, a single regular publication may lead to multiple IR publications. For instance, let us consider a window duration of 10 seconds and a window shift size of 1 second. A single matched publication can generate up to ten notifications, one for each NWR.

matched publication rate: The number of messages exchanged among brokers is significantly influenced by matched publication rate. For instance, in the above example, 20 matched publications within same 10 second window would generate only 10 notifications.

number of aggregate subscriptions: The probability of a publication to match several aggregate subscriptions increases with the total number of subscriptions. If a publication is matched by multiple subscriptions, early aggregation will generate at least one IR message for each of the matched subscriptions. In contrast, late aggregation only sends one publication through.

number of regular subscriptions: If a publication matches a non-aggregate and an aggregate subscription, both an IR and the publication itself are sent in the early approach. Higher amount of regular subscriptions increases the chance of overlap with aggregate subscription.

Most above factors except for the matched publication rate tend to favor late aggregation when the value of the corresponding parameter increases. In contrast, a high matched publication rate benefits early aggregation. As these factors dynamically change across the brokers and over time, one approach cannot outperform the other in all situations. Therefore we propose an adaptive aggregation approach which continuously monitors and weighs in these factors and switches to the appropriate aggregation mode as needed.

G. Adaptive approach

We propose a predictive solution based on a heuristic. Based on this heuristic, each broker independently decides to run either in: *Aggregate* mode that activates the IPS for each received publication or *Forward* mode, in which publications are forwarded without aggregation. If the number of publications to aggregate is greater than the expected number of notifications,the broker operates in the *Aggregate* mode, otherwise, it switches to the *Forward* mode. We follow the general adaptation methodology proposed in [23]. In each broker, we count the number of incoming messages and aggregated notifications generated during a sampling period. We then adapt the mode of the broker based on a comparison between these two numbers within this period.

An important constraint is that a publication is aggregated for all NWRs or none. This requirement is essential to support duplicate sensitive operators. Indeed, if a publication is added for some NWRs but not the others, the downstream broker is unable to infer for which NWRs it has already been aggregated, and could aggregate the same publication twice, leading to incorrect results. In order to satisfy this constraint, publications that are meant to be aggregated, might be forwarded and tagged to inform the downstream brokers that they should be aggregated.

To illustrate the behavior of our adaptive solution, we consider the example described in Fig. 3(c). In addition to an aggregate subscription, a subscriber issues a regular subscription: {sym='*IBM*', value>'*4*'}. Therefore, broker B_1 has to send publications P_1 and P_2 for the regular subscription. If B_1 aggregates these publications, it will generate 3 messages (P_1, P_2 and one intermediate result for the aggregation) when there are only 2 publications. Following our heuristic, B_1 should be in the *Forward* mode. In this example, we assume that B_1 is already in this mode, then B_3 receives publications P_1 and P_2 from B_1 and a message P_5 with the intermediate result from B_2. Assuming that B_3 is in the *Aggregate* mode, it aggregates P_1, P_2 and P_5 and generates one message P_6 with an intermediate results. Then B_3 sends P_6 along with P_1 and P_2. On the other hand, if B_3 were in the *Forward* mode, it would send the publications P_1, P_2, and P_5.

IV. Evaluation

In this section, we experimentally compare the proposed approaches. We use the total number of messages exchanged between brokers and processing time as the main metrics for comparison. We also evaluate the distribution of messages across brokers to measure the ability of each solution to balance load effectively. To this end, we consider the standard deviation of the number of messages received by different brokers.

For the sake of simplicity, we refer to the first metric as *number of messages*, the second metric as *processing load* and the third metric as *dispersion of messages* for the remainder of this section.

A. Experimental setup

We evaluate our system using two different datasets. The first dataset contains real traffic monitoring data extracted from the ONE-ITS service [19]. We generate publications out of the data produced by loop detectors that are located at major highways in Toronto. The resulting publications contain 12 distinct fields[1]. The second dataset is from stock market application (obtained from daily stock data of Yahoo! Finance) which is commonly used to evaluate pub/sub systems [24]. We used 49 stock symbols from the dataset with each stock publication containing 8 distinct fields.

We implemented our solution in Java within the framework of the PADRES[2] pub/sub system. Our deployment setup consists of a cluster of 16 servers, where 4 brokers act as core brokers forming a chain. We attach 3 edge brokers to each core broker. Out of the 12 edge brokers, 3 are exclusively dedicated to connecting publishers, 3 brokers to connecting subscribers while the remaining 6 brokers have a mix of publishers and subscribers attached. For the Traffic dataset, we consider 162 loop detectors as publishers. Additionally, all publishers which correspond to loop detectors monitoring the same direction of the same highway are connected to the same broker. For the Stock market dataset, we provide a separate publisher for each stock symbol.

For most of these experiments, we vary two parameters: the publication rate from 14 to 810 pub/s for the Traffic dataset and from 37 to 1100 pub/s for the Stock market dataset, and the number of subscriptions from 9 to 450 for both dataset. We create a mix of aggregate and regular subscriptions with varying degrees of overlap for some experiments. By default, we keep a 50% ratio of aggregate subscriptions. Each aggregate subscription is assigned a duration and a shift size of 2 seconds ($\omega = \delta = 2$). Each experiment is running for 1000 seconds, which represents 500 aggregation periods.

B. Impact of the publication rate

In this section, we evaluate the impact of the publication rate for each of our approaches.

1) Number of messages: Fig. 4(a) demonstrates that late aggregation performs better than early aggregation for low publication rates. On the other hand, when the publication rate increases, early aggregation generates comparatively fewer messages.

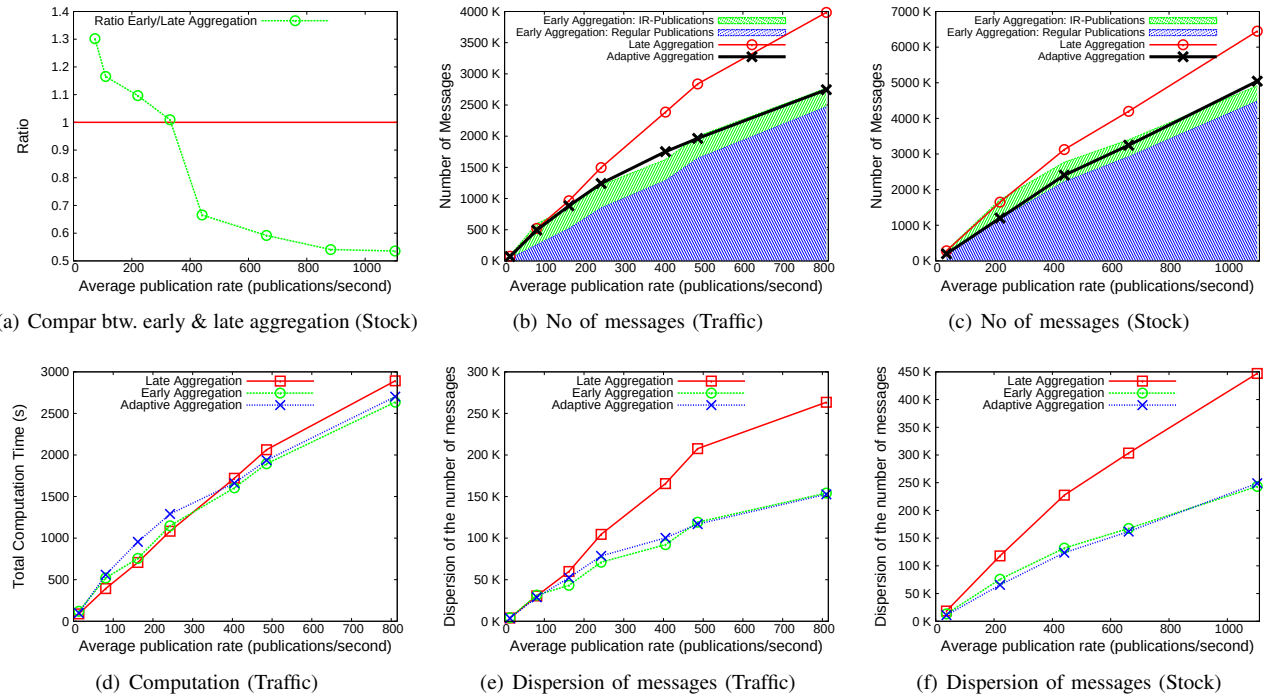
Figure 4(b) and 4(c) show the total amount of exchanged messages for various publication rates for the Traffic and Stock datasets. The two filled curves represent the number of publication messages generated by early aggregation for regular and aggregate subscriptions. The share of messages due to regular subscriptions is the same for all approaches. Therefore, Fig. 4(c) and 4(b) show the reduction of messages for aggregate subscriptions. We observe that the growth of messages slows down for early aggregation when the publication rate reaches 400 pub/s. Indeed, when all NWRs have at least one publication, we do not send additional messages when using early aggregation.

Adaptive aggregation sets brokers in the *Forward* mode when the publication rate is low. Therefore, it remains close to late aggregation (see Fig. 4(b)). Even with a low publication rate, some brokers switch to the *Aggregate* mode which leads to improved results compared to late aggregation (see Fig. 4(c)). On the other hand, with a high publication rate, all brokers switch to the *Aggregate* mode. We can notice that due to the predictive nature of adaptation, a broker may decide to switch to a mode which produces more messages for a short duration. This phenomenon can be observed in the range between 300 and 400 pub/s in Fig. 4(b).

2) Dispersion of messages: Although aggregation reduces the number of messages, this reduction may not be equally distributed amongst the brokers. Figure 4(e) and 4(f) show that dispersion is at its highest for late aggregation and lowest for adaptive aggregation. The relatively even distribution of communication load in adaptive aggregation is due to the fact that the brokers asynchronously switch to different modes at different points in the execution.

[1] http://msrg.org/datasets/traffic

[2] http://msrg.utoronto.ca/projects/padres



(a) Compar btw. early & late aggregation (Stock)

(b) No of messages (Traffic)

(c) No of messages (Stock)

(d) Computation (Traffic)

(e) Dispersion of messages (Traffic)

(f) Dispersion of messages (Stock)

Figure 4. Experimental results with varying publication rate

3) Processing load: There are three main sources of processing load common for all approaches. The first source is related to the matching of conditional predicates. This cost applies to both regular and aggregate subscriptions. The second source is due to matching of intermediate aggregation results. The last factor is the aggregation computation, which is required in all three approaches. Additionally, adaptive aggregation has a fourth overhead, namely computation of the adaptation heuristic.

When the publication rate is below 300 pub/s (see Fig. 4(d)), late and early approaches send almost the same amount of messages. Therefore, the matching cost is similar, implying that late and early aggregation have almost the same processing load. Adaptive aggregation has a slightly higher computation load because of the heuristic overhead. For higher publication rates, the amount of messages increases and the matching cost becomes higher for late aggregation.

C. Impact of the number of subscriptions

This section explores the effect of number of subscriptions (denoted as #sub) on the number of messages, processing load, and dispersion of messages using the Traffic dataset. Here, the publication rate is set to 162 per second.

1) Number of messages: Figure 5(a) presents the number of messages when varying *#sub* for all approaches with 50% of aggregate subscriptions. Two parameters have a prominent effect on the number of messages in this case: the number of overlapping subscriptions (for which publications are shared) and the average number of publications matching one NWR. If the first parameter increases, some publications are matched by more than one subscription, which causes early aggregation to send more messages than the late solution. In contrast, if the second parameter increases, early aggregation becomes comparatively better. To show this effect in the plot, we consider two different cases of *#sub*:

Below 180: The number of overlapping subscriptions is low compared to the number of publishers. From the publication rate, we can estimate that one NWR matches around 2 publications. Therefore, early aggregations performs better than late aggregation. Notably, adaptive aggregation imitates early aggregation.

Above 180: The influx of additional subscriptions increases the amount of overlap. Consequently, one publication can match several aggregate subscriptions, causing the number of messages to multiply. Therefore, early aggregation performs worse than late aggregation. This phenomenon does not occur for adaptive aggregation as some brokers decide to forward these publications. This flexibility allows the adaptive solution to send fewer messages than our baselines.

2) Processing load: As shown in Fig. 5(c), the computation time for late aggregation is minimum amongst all approaches. The reason is that predicate matching is the only source of computation for all brokers except subscriber edge brokers. For early aggregation, all brokers have all three sources of computation (see section IV-B3). When the *#sub* is high, the computation time increases proportionally with the number of messages. Up to 200 subscriptions, adaptive aggregation is mainly in the *Aggregate* mode, and hence exhibits a similar processing load. With more subscriptions,

brokers gradually switch to the *Forward* mode. The additional processing time for adaptive aggregation is due to the cost of the heuristic which depends of the *#sub*.

3) Dispersion: Fig. 5(b) shows the dispersion of number of messages amongst brokers. For late aggregation, core brokers receive publications from multiple publisher edge brokers, which skews the number of messages sent between edge and core brokers. As mentioned above for up to 180 subscriptions, a NWR typically matches 2 publications. Hence in early aggregation, the edge brokers send fewer messages to the core brokers. Moreover, the core brokers further reduce communication by merging the received IR publications, which results in lower dispersion. Above 180 subscriptions, early aggregation sends more notifications which tends to reduce the gap between early and late aggregation. Adaptive aggregation moderates the number of notifications at each broker and yields the lowest dispersion compared to the other approaches. The computation dispersion for late aggregation (see Fig. 5(d)) stays low as subscriber edge brokers perform all aggregation, which balances the load with core brokers. It is high for early aggregation because subscriber edge brokers get the least number of messages and perform less computation compared to the other brokers. In the adaptive approach, brokers switch between different modes, which keeps its dispersion between that of early and late approaches.

D. Impact of the ratio of aggregate to regular subscriptions

Figure 5(e) shows how the number of messages exchanged depends on the percentage of aggregate subscriptions. For this experiment, we consider 135 subscriptions and vary the percentage of aggregate subscriptions, 0% meaning that all 135 subscriptions are regular subscriptions and 100% meaning that all 135 are aggregate subscriptions. When the share of aggregate subscriptions increases, early aggregation reduces the amount of messages proportionally. With a mix of regular and aggregate subscriptions, adaptive aggregation achieves further reduction in messages by leveraging the overlap between subscriptions. If there are only aggregate subscriptions, adaptive aggregation performs similarly to early aggregation. In this scenario, we can also observe that late aggregation is slightly reducing the number of messages. This is due to the aggregation performed at subscriber edge brokers before delivering the results to the client.

E. Conclusion

Minimizing the number of messages forwarded by a distributed aggregation scheme in pub/sub is challenging. Depending on the publication rate and the number of subscriptions, the trends in comparing late and early aggregation can be reversed.

We observe that our adaptive aggregation effectively switches between early and late aggregation according to these factors. Moreover, it performs aggregation at intermediate brokers which results in further message reduction and more even distribution of the message load amongst brokers.

Notably, despite having additional computation overhead for the heuristic, adaptive aggregation has processing load lying between that of late and early aggregation.

V. Limitations of adaptive aggregation

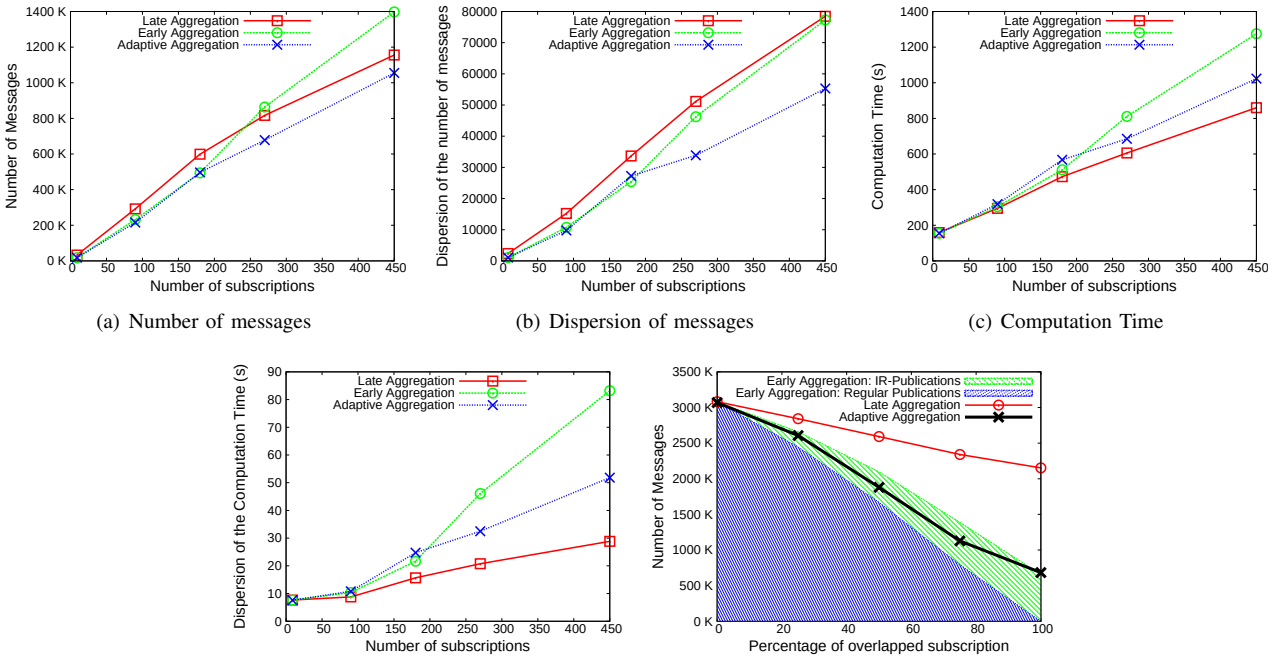
Although the experiments presented in Sect. IV show that adaptive aggregation performs generally better than late and early aggregation, there could be specific settings and corner cases where adaptive aggregation generates a higher number of messages compared to either early or late aggregation. First of all, the adaptive solution employs a predictive strategy: given an incoming publication, the broker decides where the publication should be aggregated or not by considering a history of publications received so far. As for any predictive strategy, it can be rendered ineffective by an adversary that constantly reverses the trends. Moreover, the decision is based on a specific heuristic for estimating the amount of traffic that the broker will produce in the future. A more precise heuristic may yield further traffic reduction.

In the considered adaptive solution, the broker makes a decision for an incoming publication without considering the tradeoffs on a per-subscription basis. This works well when similar amount of aggregation traffic is produced for different subscriptions. If this assumption is not valid, however, we may need to consider adaptation at the granularity of individual subscriptions. Furthermore, the decision is taken by each broker independently. A strategy coordinated across neighbor brokers may turn out to be more effective.

Finally, a broker using the adaptive solution is frequently switching between two modes: forwarding publications and aggregating them. Each switch incurs a transition period during which a broker tends to generate more messages. For example, the same publication can result in several notifications for some aggregators and in *raw* publications for others, thereby leading to a temporary increase in traffic. If a broker is experiencing frequent transition periods, the cost of those transitions may exceed the gain of adaptation. Fortunately, transitions tend to be fairly infrequent in practice, as corroborated by the evaluation in IV.

VI. Related Work

In general, distributed aggregation in data-intensive applications focuses on two aspects: processing and distributed data flows. Cayuga [14] and the approach in [15] concentrate on processing and provide an efficient query processing engine for complex event patterns which scales well in terms of the number of events and queries. Since Cayuga is centralized, Brenna et. al. [25] provides a distributed extension where the matching process spans multiple machines. On the other hand, our primary focus is on optimizing distributed data flows. Our traffic use case requires efficient distribution of both regular and aggregation subscriptions. We therefore extend pub/sub to efficiently support aggregation. Our main performance objective is to reduce the number of notifications exchanged between brokers. Note that the matching



(a) Number of messages (b) Dispersion of messages (c) Computation Time

(d) Disp. of computation (e) Impact of aggre. sub

Figure 5. Number and dispersion of messages with respect to subscriptions for Traffic dataset (540 pubs/min, 50% overlapping subscriptions)

process at each broker is self-contained as opposed to the distributed matching presented in [25]. Thus Cayuga and similar techniques are complementary to our contributions.

Meta [26] and gridStat [2] address the second challenge by using a control layer for efficient routing. However, this control layer is incompatible with elaborate routing protocols used in pub/sub, in particular it does not consider regular subscriptions. It also does not align with our goal to provide lightweight and pluggable aggregation component for existing pub/sub systems. Recent work ASIA [16] proposes to aggregate system metadata which is orthogonal to our support for content (publications) aggregation.

Aggregation in distributed pub/sub systems shares some common challenges with distributed stream processing systems (DSPS). This includes aggregating data from distributed publication sources as well as processing and serving multiple queries [27]. Publication sources are a priori known in DSPS systems before the query plans are computed and the placement of operators is determined [28]. In particular, computing a query plan requires a global view of the overlay topology. Conversely in pub/sub systems, the location of sources and sinks are changing over time and broker visibility is restricted to a neighbourhood, which prohibits the use of static query plans. Moreover, source nodes in a DSPS environment are assumed to continuously produce data for long durations while, in pub/sub, publishers generate events intermittently at arbitrary points in time making query planning optimizations ineffective for pub/sub. However, some of the optimization techniques from DSPS such as multi-query optimization [22] can be utilized in pub/sub. For example, the notification schedule synchronization technique in [29] that optimizes message traffic by batching is orthogonal to and compatible with our approach.

A number of distributed schemes for disseminating aggregate information over unstructured P2P overlays have been proposed in the literature [4], [30], [31], [18], [32], [3]. These approaches can be extended for content-based pub/sub dissemination, at the cost of scalability issues [33]. As an alternative to integrating aggregation with pub/sub, a standalone aggregation engine external to the pub/sub layer could be used. However, as mentioned in ASIA [16], it can hamper QoS for regular publications. Most of standalone approaches create their own aggregation and distribution trees for efficient computation and dissemination [28]. These aggregation trees may not always align with the routing overlay created by pub/sub. Therefore, as the same set of publications may match both regular and aggregate subscriptions, an additional overhead is incurred by sending redundant messages through two different dissemination layers.

VII. Conclusions

Modern use cases for pub/sub require stream processing capabilities that are not natively supported by traditional solutions. Although aggregation can be provided through the end-to-end principle, late aggregation at the subscriber edge is not practical with data intensive applications where disseminating the entire event stream through the pub/sub system is prohibitively expensive. To alleviate this problem, we propose a distributed approach to pub/sub aggregation. We also introduce mechanisms integrated with the pub/sub

system that provides aggregation of publications within the broker overlay, which reduces the amount of in-network traffic. We consider an early aggregation approach that places aggregators at the publisher edges. This approach significantly reduces traffic in workloads where the publication rate is higher than the aggregation notification rate. However, we demonstrate that this approach is not optimal, and propose an adaptive rate-based solution which outperforms both baselines using real datasets extracted from our traffic monitoring use case.

REFERENCES

[1] A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. S. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts, "Linear road: a stream data management benchmark." VLDB Endowment, 2004, pp. 480–491.

[2] S. F. Abelsen, H. Gjermundrd, D. E. Bakken, and C. H. Hauser, "Adaptive data stream mechanism for control and monitoring applications," in *Computation World*, 2009.

[3] M. Jelasity, A. Montresor, and O. Babaoglu, "Gossip-based aggregation in large dynamic networks," *ACM Trans. Comput. Syst.*, vol. 23, no. 3, pp. 219–252, 2005.

[4] P. Jesus, C. Baquero, and P. S. Almeida, "A survey of distributed data aggregation algorithms," University of Minho, Tech. Rep., 2011.

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

[6] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM Comput. Surv.*, vol. 35, no. 2, pp. 114–131, 2003.

[7] C. Schuler, H. Schuldt, and H.-J. Schek, "Supporting reliable transactional business processes by publish/subscribe techniques," in *Proc. of TES*, 2001, pp. 118–131.

[8] G. Cugola, E. Di Nitto, and A. Fuggetta, "The JEDI event-based infrastructure and its application to the development of the opss wfms," *IEEE Trans. SE*, vol. 27, no. 9, 2001.

[9] T. Fawcett and F. Provost, "Activity monitoring: noticing interesting changes in behavior," in *Proc. of SIGKDD*, ser. KDD '99. ACM, 1999, pp. 53–62.

[10] Y. Tock, N. Naaman, A. Harpaz, and G. Gershinsky, "Hierarchical clustering of message flows in a multicast data dissemination system," in *Proc. of PDCS*, 2005, pp. 320–326.

[11] I. Rose, R. Murty, P. Pietzuch, J. Ledlie, M. Roussopoulos, and M. Welsh, "Cobra: content-based filtering and aggregation of blogs and RSS feeds," in *Proc. of NSDI*.

[12] I. Koenig, "Event processing as a core capability of your content distribution fabric," in *Gartner Event Processing Summit*, 2007.

[13] V. Setty, G. Kreitz, R. Vitenberg, M. van Steen, G. Urdaneta, and S. Gimåker, "The hidden pub/sub of spotify (industry article)," in *Proc. of DEBS*, 2013.

[14] A. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. White, "Towards expressive publish/subscribe systems," in *Proc. of EDBT*, 2006, pp. 627–644.

[15] J. Sventek and A. Koliousis, "Unification of publish/subscribe systems and stream databases: the impact on complex event processing," in *Proc. of Middleware*, 2012, pp. 292–311.

[16] S. Frischbier, A. Margara, T. Freudenreich, P. Eugster, D. Eyers, and P. Pietzuch, "ASIA: Application-specific Integrated Aggregation for publish/subscribe middleware," in *Proc. of Middleware*, 2012, pp. 6:1–6:2.

[17] B. Chandramouli and J. Yang, "End-to-end support for joins in large-scale publish/subscribe systems," *Proc. VLDB Endow.*, vol. 1, no. 1, pp. 434–450, 2008.

[18] R. van Renesse and A. Bozdog, "Willow: DHT, aggregation, and publish/subscribe in one protocol," in *Proc. of IPTPS*, 2004, pp. 173–183.

[19] "ONE-ITS Online Network-Enabled Intelligent Transportation Systems." [Online]. Available: http://one-its-webapp1.transport.utoronto.ca

[20] A. Koulakezian and A. Leon-Garcia, "CVI: Connected Vehicle Infrastructure for ITS," in *Proc. of PIMRC*, 2011, pp. 750 –755.

[21] S. Biswas, R. Tatchikou, and F. Dion, "Vehicle-to-Vehicle wireless communication protocols for enhancing highway traffic safety," *IEEE comm. mag.*, vol. 44, no. 1, pp. 74–82, 2006.

[22] S. Krishnamurthy, C. Wu, and M. Franklin, "On-the-fly sharing for streamed aggregation," ser. SIGMOD '06, 2006.

[23] M. Salehie and L. Tahvildari, "Self-adaptive software: Landscape and research challenges," *ACM Trans. Auton. Adapt. Syst.*, vol. 4, no. 2, pp. 14:1–14:42, 2009.

[24] A. K. Y. Cheung and H.-A. Jacobsen, "Publisher placement algorithms in content-based publish/subscribe," *Proc. of ICDCS*, pp. 653–664, 2010.

[25] L. Brenna, J. Gehrke, M. Hong, and D. Johansen, "Distributed event stream processing with non-deterministic finite automata," in *Proc. of DEBS*, 2009.

[26] M. Wood and K. Marzullo, "The design and implementation of meta," in *Reliable Distributed Computing with the ISIS Toolkit*, 1994, pp. 309–327.

[27] N. Jain, D. Kit, P. Mahajan, P. Yalagandula, M. Dahlin, and Y. Zhang, "STAR: Self-tuning aggregation for scalable monitoring," in *VLDB*, 2007, pp. 962–973.

[28] P. Yalagandula and M. Dahlin, "Shruti: A Self-Tuning Hierarchical Aggregation System," in *Proc. of SASO*. IEEE Computer Society, 2007, pp. 141–150.

[29] L. Golab, K. G. Bijay, and M. T. Özsu, "Multi-query optimization of sliding window aggregates by schedule synchronization," in *Proc. of CIKM*, 2006, pp. 844–845.

[30] R. Van Renesse, K. P. Birman, and W. Vogels, "Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining," *ACM Trans. Comput. Syst.*, vol. 21, no. 2, pp. 164–206, 2003.

[31] P. Yalagandula and M. Dahlin, "A scalable distributed information management system," *SIGCOMM Comput. Commun. Rev.*, vol. 34, no. 4, pp. 379–390, 2004.

[32] T. Repantis and V. Kalogeraki, "Hot-spot prediction and alleviation in distributed stream processing applications," in *Proc. of DSN*, 2008, pp. 346–355.

[33] R. Baldoni, L. Querzoni, S. Tarkoma, and A. Virgillito, "Distributed event routing in publish/subscribe systems," in *MNEMA*. Springer, 2009, pp. 219–244.