

Subscription Summarization: A New Paradigm for Efficient Publish/Subscribe Systems¹

Peter Triantafillou
Research Academic Computer Technology
Institute, and University of Patras
Patra, 26500, Greece
peter@ceid.upatras.gr

Andreas Economides
Department of Electronic and Computer
Engineering
Technical University of Crete
Chania, 73100, Greece
oikand@softnet.tuc.gr

Abstract

We contribute a new paradigm for publish/subscribe systems. It is centered on the novel notion of subscription summarization. We first present the summarization structures for a broker's subscriptions and accompanying algorithms, which operate on the summary structures to match incoming events to the brokers with relevant subscriptions and for the maintenance of subscriptions in the face of updates. Second, we present novel algorithms for efficiently propagating subscription summaries to brokers. Finally, we present a novel algorithm for the efficient distributed processing of incoming events, utilizing the propagated subscription summaries to route the events to brokers with matched subscriptions. We study the performance of our contributions, comparing them against a baseline approach and against corresponding techniques employed in a well-known event-based distributed system. Our results show the significant performance gains introduced for both the subscription propagation and distributed event processing tasks.

1. Introduction

Event-based, publish/subscribe, systems are receiving increasingly greater attention as a means to develop large-scale data retrieval and dissemination systems that enable personalized data delivery. In these systems a different model is adopted for finding and delivering interesting distributed information to the distributed users: a user declares his interests and receives the appropriate information/events, as the events matching these interests take place. Such a system gives users the ability to receive information dynamically at the time it becomes available.

Publish/Subscribe (pub/sub) systems therefore connect information providers and consumers delivering personalized information.

System Architecture

A pub/sub system is comprised of three main entities. A consumer, who subscribes his interests to the system, a provider, who publishes events and the pub/sub infrastructure which has the responsibilities to (i) match each event to all related subscriptions and (ii) to deliver the matching events to the corresponding consumers. The architecture of a pub/sub system (figure 1) consists of:

- One or more **Event Sources** (ES) / Producers. An Event Source produces events, say, in response to changes to a real world variable that it monitors.
- An **Event Brokering Network** (EBN). The events are published to the Event Brokering Network, which matches them against a set of subscriptions, submitted by users (consumers) in the system.
- One or more **Event Displayers** (ED) / Consumers. If a user's subscription matches an event, it is forwarded to the Event Displayer for that user. The Event Displayer is responsible for alerting the user.

The paper is organized as follows. Section 2 briefly reviews related work, defines the research problem, and outlines our contributions. We develop the summary data structures and the associated algorithms in section 3. Section 4 presents the notion of multi-broker summaries, their maintenance, the associated algorithm for propagating subscriptions among the brokers of the system, and algorithms for the distributed event processing, routing events to interested brokers. In section 5 we present our performance study and the results showing the significant benefits our approach introduces. Section 6 concludes the paper.

¹ This research was supported by the European Commission's IST/FET DBGLOBE project and Integrated Project DELIS.

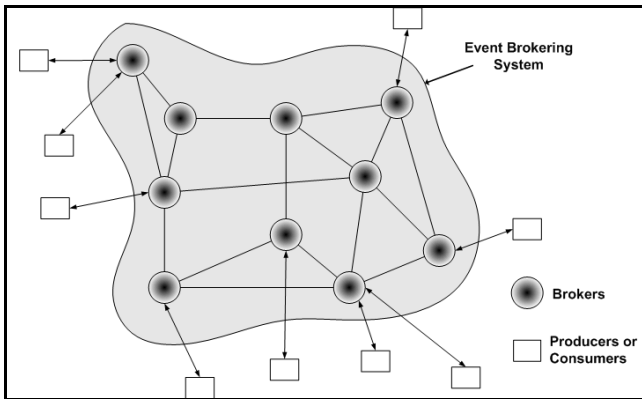


Figure 1. A distributed architecture of a pub/sub system.

2. Research perspective

2.1. An overview of related work

The first pub/sub systems were based on the concepts of group (also known as channel-based systems) or subject (a.k.a. topic-based systems). Channel-based systems [12], [19] categorize events into pre-defined groups. Users subscribe to the groups of interest and receive all events for these groups. In subject-based systems [7], [14] each event is enhanced with a tag describing its subject. Subscribers can declare their interests about event subjects flexibly using string patterns. In the content-based model [3], [5], subscribers use flexible querying languages to declare their interests with respect to the contents of the events. For example, such a query could be "give me the price of stock A when the price of stock B is less than X". A slightly different model than this is the "content-based with patterns" model [6], [13], with extra functionality on expressing user interests.

This area has enjoyed considerable attention. Examples of this research's results are the Gryphon [1], [2], Siena [6], Jedi [7], Le Subscribe [8], Ready [10], Rebecca [11], Hermes [15], and Elvin [17] systems. In the commercial world, several systems have been implemented, with some prominent examples being the CORBA Event service [12], CORBA Notification service [13], iBus [18], Jini [19], Tibco [20] and Vitria [22].

Event and Subscription Types

We employ the event and subscription schemata found in the literature [6], (our paradigm can also be used with any other schema with similar attribute types and operators).

Event Schema

The Event Schema of this model is an untyped set of typed attributes. Each attribute consists of a type, a name

and a value. The *type* of an attribute belongs to a predefined set of primitive data types commonly found in most programming languages. The attribute's *name* is a string, while the value can be in any range defined by the type. The whole structure of type – name – value for all attributes constitutes the event.

Event 1		
Type	Name	Value
string	exchange	= NYSE
string	symbol	= OTE
date	when	= Jul 1 12:05:25 EET 2003
float	price	= 8.40
integer	volume	= 132700
float	high	= 8.80
float	Low	= 8.22

Figure 2: An event example.

Subscription Schema

The subscription schema is more general, allowing for expressing a rich set of subscriptions, containing all interesting subscription-attribute data types (such as integers, strings, etc.) and all interesting operators (=, ≠, <, >, prefix ">*", suffix "*<", containment "*", etc.).

An event matches a subscription if and only if all the subscription's attribute constraints are satisfied. A subscription can have two or more constraints for the same attribute. Finally, an event can have more attributes than those mentioned in the subscription attributes.

Subscription 1			Subscription 2		
Type	Name	Constraint	Type	Name	Constraint
string	exchange	N*SE	string	symbol	>* OT
string	symbol	= OTE	float	price	= 8.20
float	price	< 8.70	integer	volume	> 130000
float	price	> 8.30	float	low	< 8.05

Figure 3: Two subscription examples.

2.2. Motivations and problem definition

We consider pub/sub systems in which events of a producer attached to any broker may be of interest to any subscriber, attached to any broker. When designing and implementing large-scale, pub/sub systems, one must guarantee the efficiency of the system during both of its (concurrent) phases of operation. The first phase is the propagation of subscriptions among the (potentially large numbers of) brokers. Brokers require this information in order to forward incoming events only to interested users, filtering out unrelated events, which can save significant overheads (i.e., network bandwidth and processing time).

The second phase is the event processing phase, during which produced events are either filtered out, or matched and then routed to brokers with subscriptions matching the events.

In this paper, we will compare our approach against key notions and algorithms employed by the Siena system. We chose Siena since it is a well-established, well-

regarded system. A fundamental notion in Siena is that of *subscription subsumption*, utilized extensively during subscription propagation and during event routing. Briefly, an attribute-value constraint of a subscription is said to be subsumed by that of another subscription if the values are the same (in the case where the desired attribute constraint is specified using the equality operator) or if it is contained (say, when it is specified using the prefix/suffix/containment operators). A subscription is said to be subsumed by another, if all attribute constraints of the former are subsumed by the attribute constraints of the later.

Siena's subscription propagation and event routing is based on such subsumption relations among subscriptions. A subscription is not forwarded by a broker to another broker if the former has already forwarded to the latter a subscription that subsumes this one. Matched events follow the paths setup by subscriptions. (Siena also uses the notion of advertisements. However, here we will only consider the subscription propagation and event routing as they are affected by the notion of subsumption).

2.3. Contributions

We contribute the notion of per-broker subscription summaries, compacting subscription information. The summarization mechanism supports event/subscription schemata that are rich with respect to the attribute types they include (i.e., numeric types, ranges, strings) and powerful with respect to the operators for these attributes (i.e., $<$, $>$, $=$, \neq , prefix, suffix, string containment, etc).

The novelty of the proposed paradigm is that it is *subscription-summary-centric*, as opposed to being subscription-centric. Each incoming subscription is dissolved into its attribute-value pairs, which are in turn merged into their corresponding summary structures. In this paradigm there are no subscription entities, only subscription summaries. The summaries will ensure performance benefits with respect to (i) the network bandwidth required to propagate subscriptions among brokers and (ii) the storage overhead to maintain them.

On the other hand, new algorithms are needed to operate on these summaries during both phases of operation of a pub/sub system. Thus, we develop the accompanying event matching algorithm. Our matching algorithm complexity analysis shows that, we can ensure the same complexity as competing approaches.

Finally, we contribute the notion of multi-broker subscription summaries and develop a distributed algorithm for efficiently propagating multi-broker summaries among brokers. This algorithm ensures further network bandwidth savings and low hop counts (i.e. path lengths) among brokers for subscription propagation.

With respect to the event processing phase, we develop distributed algorithms for routing events to the interested brokers, which ensure low hop counts among brokers for the routing of events, to all interested brokers.

We present a performance study. The results clearly show that the proposed paradigm significantly improves the performance of a pub/sub system, in terms of network bandwidth for subscription propagation, storage, and number of brokers involved in subscription propagation and event processing.

These contributions build on our initial attempt towards a subscription-summary-centric solution for pub/sub systems [21]. In comparison, in this paper we present improved summary structure, matching algorithms, algorithms for their maintenance, algorithms for creation, propagation, and maintenance of multi-broker summaries, and a detailed performance evaluation.

3. Per-broker subscription summaries

In this section we develop the set of data structures representing the summarized subscriptions received at a broker. It is this compacted, summarized, subscription information that is propagated to the other brokers. We also describe the algorithm that operates on the summaries to match/filter incoming events at a broker.

We assume that: (i) a named attribute cannot have two different data types, (ii) the number of attributes in the system are predefined, as well as the specification of these attributes (name – type), and (iii) the set of supported attributes is ordered and known from each broker.

3.1. Data structures and operators

Per-broker subscription summaries consist of two data structures, one for arithmetic and one for string attributes.

Arithmetic Attribute Constraint Summaries (AACS)

For each different arithmetic attribute, a broker is implementing an *AACS* consisting of two arrays. $AACS_{SR}$ is an array with two columns and a variable number (n_{sr}) of rows. Each row represents non-overlapping sub-ranges of values specified in subscriptions for the specific attribute. The first (second) column shows the lower (upper) bound of such a sub-range. The second array, $AACS_E$ is used when an arithmetic constraint in a subscription has an equality operator for a value that is not included in the existing sub-ranges. It has only one column and a variable number of rows (n_e) with the same meaning as in the first array. In both arrays, each row has a list with the subscription ids that have a constraint that is satisfied by the row's value(s).

For every new/incoming constraint of an arithmetic attribute, if it is not included in the existing sub-ranges or

equality values, a new row is added for it. Moreover the id of the new subscription that the constraint belongs to, is added to the subscription id list for the specific row.

AACCS for attribute <i>price</i>			
Range	min	max	id list
	8.30	8.70	→ S1
Equality	value	id list	
	8.20	→ S2	

Figure 4: An AACCS example.

String Attribute Constraint Summaries (SACS)

SACS holds information about the constraints of subscriptions' string attributes. For each different string attribute, a broker is implementing a SACS structure using an array of values. Each row in the array represents a general constraint that may cover (i.e., subsume) one or more of the existing constraints. For example the constraint "m*t" can cover constraints like "microsoft" or "micronet", etc. If a more general constraint appears then the current is substituted by the new one. A linked list of subscription ids is kept in each row, as in AACCS.

For every constraint of the same string attribute, a new row is added if it not covered by some existing one. Also, the id of the subscription to which the constraint belongs is added to the id list of the specific row. In the case that the new constraint is covered by an existing one then we just add its subscription id in the appropriate list.

SACS for attribute <i>symbol</i>	
>*OT	→ S1, S2

Figure 5: A SACS example.

3.2. Subscription ids

A subscription id is the concatenation of three parts:

1. c_1 : The id of the broker receiving the subscription (i.e., where the subscription "belongs"). The size of this field in bits is equal to the rounded-up base-2 logarithm of the total number of brokers in the system (e.g. in a system with 1000 brokers, c_1 would be 10-bits long).
2. c_2 : The id of the subscription itself. The size of this field in bits is equal to the rounded-up base-2 logarithm of the maximum number of outstanding subscriptions a broker can have (e.g. if each broker needs to manage 1,000,000 of subscriptions, c_2 would be 20-bits long).
3. c_3 : The ids of the attributes that comprise the subscription. The size of this field in bits is equal to the total number of attributes supported by the system. Each subscription attribute is represented in c_3 by one bit. Thus, for each attribute for which there exists a constraint in the specific subscription, the corresponding bit is set to 1.

Assume an example system having 4 brokers, each of which can support 8 outstanding subscriptions, with an attribute schema including 7 attributes. The subscription id depicted in figure 6, identifies subscription 1 ($c_2=1$), belonging to broker 2 ($c_1=2$), comprised of constraints on attributes with ids of 3, 5, and 6 (right to left).

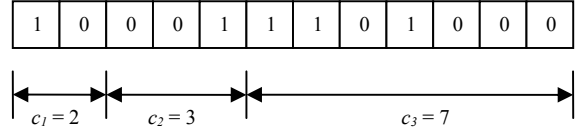


Figure 6: An example subscription id.

3.3. The matching algorithm

For simplicity of presentation we consider two brokers, *A* and *B*. Assume broker *A* receives a number of subscriptions, constructs the corresponding subscription summaries and sends them to broker *B*, which will use them to identify all events of interest to *A*. The algorithm for matching an incoming event at broker *B* against the subscriptions of broker *A* is as follows:

1. **For** every attribute in the incoming event, scan the summary data structures for satisfied subscription constraints on the attributes of the event, and collect the corresponding subscription-id lists. If no constraint is satisfied then, obviously, the specific event doesn't correspond to any subscription of broker *A*:
 - If the attribute is arithmetic, find if there is a match using the corresponding AACCS structure(s) and the **Check_for_a_value_match(type arithmetic)** method.
 - If it is a string attribute, find if there is a match using the SACS structure(s) and the **Check_for_a_value_match(type string)** method.
 - Keep a counter for every subscription id in the collected lists of the number of lists this subscription id is found to belong.
2. **For** every unique subscription id that has been collected, check if all of its associated attributes (using c_3) appear in the satisfied attributes of step 1, by checking whether the corresponding counter (from step 1) is equal to the number of its associated attributes.
 - If they are equal, then there is a match and the subscription id is kept in order to inform broker *A*.
3. **If** at least one subscription id is kept, then the event, and all the subscription ids that have been matched by this event, are forwarded from broker *B* to broker *A*.

Algorithm 1: Matching an event at broker *B* against the subscription summaries of broker *A*.

Check_for_a_value_match (type arithmetic)

Given the attribute value, broker *B* first checks if the attribute's value belongs to any range defined in the corresponding AACCS data structures it scans the array holding the ranges, examining the min and max columns.

If so, we have a match for the attribute. Else, it continues scanning the second array (with subscribed values outside the ranges) searching for a match among the values there.

Check_for_a_value_match (type string)

In this case, the value of the attribute is checked against the corresponding *SACS* data structure(s). It follows a similar procedure to that for arithmetic attributes: For each row of the array in *SACS*, it checks if the value of the attribute in the event is covered by the constraint of the row. If none of the rows cover the value of the attribute then there is no match.

Example 1. Let us suppose that broker *A* has the subscriptions of figure 3 and the event in figure 2 is generated at *B*. First the algorithm will collect all the id lists in which the values of the attributes of the event satisfy the corresponding constraints.

To do this, the algorithm will check if there is a match in its *SACS* for the value *NYSE* of attribute *exchange*. The algorithm will find a match and collect the corresponding subscription-id list. The same procedure is also executed for the attribute *symbol* (see figure 4). Then the algorithm continues with the value of the attribute *price* (8.40). Since it is an arithmetic type, it looks at the corresponding *AACS* to find if this value is in a defined range or if it matches any of the values outside the ranges. Here we see that the range constraint "8.30 – 8.70" is satisfied. The same happens with attribute *volume* (132700), since it satisfies the constraint ">130000". Thus, we get the following subscription-id lists: *exchange* → **S1**, *symbol* → **S1**, **S2**, *price* → **S1**, *volume* → **S2**.

The unique subscriptions in this example are **S1** and **S2** with corresponding counters 3 and 2 respectively. First, the algorithm examines part c_3 of **S1** and finds that this subscription is comprised of 3 attributes. So, **S1** is kept in order to inform broker *A* about the match. However, **S2**'s c_3 is comprised of 4 attributes while its counter is equal to 2, so the event examined is of no interest to the corresponding subscriber.

Because there is at least one subscription that satisfies the event, broker *B* sends the specific event to *A* with all the subscription ids (**S1**) that are matched by the event. ■

4. Multi-broker subscription summaries

4.1. Overview of the approach

How to construct multi-broker summaries

Observe that once the subscription summaries of a set of brokers have been received at some broker *B*, it is straightforward for *B* to merge all the received summaries with its own. Values for the same string and numeric attributes are simply merged by taking the union of the corresponding sets. Also, the merging of supporting

structures like arithmetic ranges is also straightforward. Because of space limitation a detailed discussion for maintaining the summaries is omitted.

How to filter and route events

Brokers develop multi-broker subscription summaries from their neighbors. Each broker B_i creates as before its own summary. This summary is then propagated to its neighbor(s). Each such neighbor, B_j , using the same structures, merges its own subscription summary with that received from B_i (and from all of its other neighbors) and further forwards the merged (multi-broker) summary to its neighbor(s). When an event arrives at a broker, *B*, the filtering algorithm runs against the multi-broker subscription summaries received from *B*'s neighbors.

Note that, in addition to the benefits owing to the significantly faster matching/filtering of events at each broker, the actual network bandwidth required to exchange subscriptions among brokers is also reduced.

The whole process is logically divided into two phases. The first encompasses the subscription summary propagation, and the second phase encompasses the distributed processing of events at brokers.

4.2. Subscription summary propagation

The subscription summaries of brokers are propagated to the other brokers in the system, exploiting the underlying broker (overlay) topology. This phase starts at specific points of time (e.g., periodically) engaging the brokers. The process includes a number of iterations equal to the maximum degree (defined as the maximum number of neighbors a broker has) among all brokers.

At each iteration, each broker having a degree equal to the number of the current iteration, constructs a merged subscription summary, which includes its own and all subscription summaries that it received in the previous iterations from its neighbors. In addition, with each merged summary, the broker associates the set *Merged_Brokers*, which consists of the ids of the brokers whose subscriptions are included in the merged summary. If no summaries were previously received (that is the case in the first iteration) then the constructed multi-broker summary contains only this broker's subscriptions and the set *Merged_Brokers* contains only this broker's own id.

After that, each broker sends the merged summary and its *Merged_Brokers* set to a neighbor with which it has not communicated in any of the previous iterations (i.e., the neighboring brokers with equal or higher degree). If a broker has two or more such neighbors, then only one is selected (preferably the one with the smallest degree) for sending the summaries. Also, each broker stores the updated value of *Merged_Brokers*.

The propagation algorithm pseudocode is given below.

```

For  $i=1$  to MAX_DEGREE
  Each node of degree  $i$ :
  1. Merges its summary with all received summaries
    and updates the set Merged_Brokers.
  2. Sends the merged summary and the set
    Merged_Brokers to a selected neighbor with
    equal or higher degree.
End

```

Algorithm 2: Subscription Summary Propagation.

4.3. Distributed event processing: routing and filtering

When an event arrives at a broker, B , the broker need only examine the overall merged subscription summary that it keeps, checking for a match, using the matching algorithm of section 3.3. When a match is detected, the broker routes the event to the broker(s) that own the matched subscription(s), using the field c_i of each matched subscription id. In parallel, the broker also updates *BROCLLe* (Broker Check List for a specific event e) carried by each event. This list shows the brokers whose subscriptions have already been examined for a match against this event. This list is maintained by having each broker, receiving the event, add to this list the value of the set *Merged_Brokers* (i.e., its own id as well as the ids of all brokers whose subscriptions have been included in the merged summaries B has received). If *BROCLLe* does not contain all brokers, the search for a match continues, forwarding the event. The pseudocode for the distributed event-processing algorithm follows:

```

For each incoming event at a broker do
  1. Check the local merged summary for a match
  2. Update the list BROCLLe
  3. If a match is found then
    Send the event to the matched broker
  4. If BROCLLe does not contain all brokers then
    Find the broker with the highest degree whose id
    is not in BROCLLe and send it the event

```

Algorithm 3: Event routing and filtering.

There are a number of alternatives for selecting a broker to which to forward the event, which enjoy comparative performance advantages which trade-off event processing time with load distribution among brokers for matching events. In the alternative presented here, B forwards the event to the broker with the greatest degree, among the brokers whose ids are not in *BROCLLe*. It also sends to it the updated *BROCLLe*. This process is executed at each broker that receives an event together with the *BROCLLe* and stops when the broker examining the event finds that the *BROCLLe* list includes all brokers in the system.

Example 3. Consider a system with 13 brokers interconnected with the overlay topology shown in figure 7. (A tree topology is used for easier presentation, since the extra edges in a graph complicate the presentation).

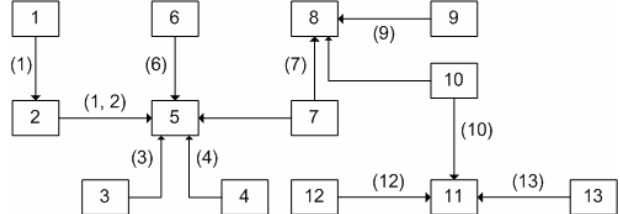


Figure 7: An example event routing and filtering.

We can see that the highest degree of a broker is 5 (node 5). In the 1st iteration the brokers 1, 3, 4, 6, 9, 12, and 13 send their subscription summaries to their neighbors. The labels on the edges show the values of the *Merged_Brokers* set. In the 2nd iteration, brokers of degree two (i.e., brokers 2, 7, and 10), construct a merged summary from their own summaries and the ones received in all previous iterations (if any). In our example, in this iteration broker 2 will create a merged summary from its own summary and the one from broker 1.

Subsequently, broker 2 sends the merged summary to broker 5, broker 7 sends its own to 5 (or 8), and broker 10 to 8 (or 11). In the 3rd iteration, brokers 8 and 11 merge the received summaries. Broker 8 will merge its own summary with the summaries received from its neighbors in all previous iterations (i.e., the summaries of brokers 7, 9 and 10). In iteration 4, there is nothing to do because there is no broker of this degree. In the last iteration, broker 5 will merge the summaries from brokers 2, 6, 3, and 4. Phase 1 (the subscription propagation phase) ends at this point. Note here that, finally, broker 5 for example will have knowledge of the summaries of brokers 1 to 6.

Suppose now that an event matching brokers 4, 8 and 13, arrives at broker 1. Broker 1 will first check (unsuccessfully) if the event matches its own summary and then will update the *BROCLLe* list adding itself to it. Then it will forward the event and the updated *BROCLLe* list to the nearest highest-degree broker, which is not included in *BROCLLe*. This is broker 5. Broker 5 in turn will check for a match in his merged summaries and will update *BROCLLe* adding all the brokers from 2 to 6 (broker 1 is already in). The check against the summary sent by broker 4 will be successful and, thus, broker 5 will forward the event to broker 4. Simultaneously, because *BROCLLe* does not yet include all brokers, broker 5 will send the event with the updated *BROCLLe* to broker 8 (since 8 is the broker with the greatest degree among those not included in *BROCLLe*). Broker 8 will find the local match and behave similarly to broker 5 and will forward the event to 11. After the match is found for broker 13, the process will end because 11 will see that all brokers are now included in *BROCLLe*. ■

5. Analysis of performance

In this section, we present an analysis of the performance of our approach. Our primary performance metrics are (i) the network bandwidth requirements, (ii) the storage requirements for subscriptions and summaries, (iii) the required number of hops between brokers for subscription propagation, and (iv) the number of hops required for the routing of events to matched brokers.

The network bandwidth is measured as the size (in bytes) a broker exchanges with all the others. The key parameters are listed in table 1.

Table 1. Parameter definition.

Symbol	Description
n_t	Total number of attribute names in the event/subscription type.
S	Average number of outstanding subscriptions in a broker.
σ	Average number of new per-broker arrived subscriptions in each period.
n_{as}	Average number of different arithmetic attributes in a subscription of a broker.
n_{sr}	Per arithmetic attribute: number of rows in $AACS_{SR}$.
n_e	Per arithmetic attribute: number of rows in $AACS_E$.
L_a	Per arithmetic attribute: size of the subscription id list.
n_{ss}	Average number of different string attributes in a subscription of a broker.
n_r	Per string attribute: number of rows in $SACS$.
L_s	Per string attribute: size of the subscription id list.
ssv	Average size of a string value (one byte per character).
sst	Storage size type of an arithmetic attribute.
sid	Storage size of a subscription id.
E	Average number of incoming events at a broker.
n_{ae}	Average number of different arithmetic attributes in an event.
n_{se}	Average number of different string attributes in an event.

5.1. The network bandwidth analysis

The following equations show the total size (in bytes) for the ‘summarizing’ data structures. We calculate the size of the structure itself plus the size of the subscription ids.

Size of the $AACS$

$$AACS = \sum_{i=1}^{n_{as}} \left((2n_{sr_i} + n_{e_i}) \times sst_i \right) + (L_{a_i} \times sid) \quad (1)$$

The first part of (1) is for calculating the sizes of the two arrays. Factor 2 is due to the two columns (min, max),

needed by each sub-range. The second part is for finding the size needed to store the subscription ids in both arrays.

Size of the $SACS$

$$SACS = \sum_{i=1}^{n_{sr}} (n_r \times ssv) + (L_{s_i} \times sid) \quad (2)$$

As in (1), the first part is for calculating the size of the array. The second part is needed for calculating the required space for the lists of the subscription ids.

So, the total bandwidth per broker, TB , is equal to the sum of the size of the two data structures (1) and (2).

5.2. Performance results

We have comparatively evaluated our approach against a baseline approach where all brokers broadcast their subscriptions to all, and against the Siena subsumption-based algorithm. We did not compare against the advertisement mechanism of Siena, since this mechanism can be employed by our system as well and since our goal is to show what performance gains are inherent in our summary-centric approach. Table 2 presents the parameter values used. We assume that all n_t appear at least once in some subscription. The ‘average’ subscription or event used in our analysis includes $n_t/2$ attributes, with 40% (60%) being arithmetic (strings). The average size of a subscription/event is 50 bytes.

Table 2. Values of parameters used.

Symbol	Values
S	= 1000 (per broker)
Σ	= 10, ..., 1000
<i>Subscription Subsumption probability</i>	= 0.1, 0.25, 0.5, 0.75, 0.9
n_t	= 10
n_{sr}	= 2
sst, sid	= 4
ssv	= 10

Recall, that some of the values for arithmetic and string attributes are subsumed. In arithmetic attributes, all subsumed values fall into the n_{sr} ranges of the attribute. The non-subsumed values are represented as different values (specified with equality operators outside the ranges). The results for configurations with the other parameter values are similar.

5.2.1. The Performance of Subscription Propagation

Tested Topologies

We have used a number of real and artificial topologies. We show the results (which are similar in all cases) assuming our brokers are organized in an overlay network topology like that of the backbone network of U.S. Cable and Wireless plc [4], having 24 nodes. This topology is representative of the performance of our

system when being included in networks such as single-ISP Content Delivery Networks, for example those by Cable and Wireless and AT&T, which number from 20 to 33 backbone nodes. Furthermore, these broker overlay (ISP backbone) networks have the property that the topologies are slowly changing, and when they do, the network nodes can be informed of the new changes. Thus, our subscription id structure does not pose a limitation.

Network Bandwidth Performance

We measured the network bandwidth requirements, as the cost for all brokers to propagate their subscriptions in a period. This cost for the baseline approach is measured as $(brokers - 1) \times average\ numbers\ of\ hops\ (from\ any\ broker\ to\ any\ other) \times brokers \times \sigma \times average\ size\ of\ a\ subscription$. For Siena's subsumption-based, subscription propagation, we used a varying probability of subsumed subscriptions. In particular, at each broker B , with a probability equal to the subscription subsumption probability, B did not forward each subscription it received to each of its neighbors. This was repeated for all brokers, until each subscription eventually reached all its destinations or was subsumed. With every result we state the corresponding subsumption probability.

In our model, not all brokers have the same subsumption probability; brokers with higher connectivity will enjoy higher probabilities, given the greater number of subscriptions that pass through them. Thus, the stated subsumption probability refers to the maximum probability among brokers. Each broker's subsumption probability is being determined as the maximum subsumption probability times the fraction of this broker's degree over the maximum degree.

For our approach the subscription summary sizes differ between brokers. Brokers that merge summaries received from other brokers, have greater summary sizes. The appropriate size is determined as follows: brokers that merge summaries from n other brokers, appropriately increment the number of distinct values expected to store in the merged summary and define it to be the sum of the distinct values at each of these n brokers. In general, the sizes of the summary data structures that are sent by a broker are based on (i) the value of σ , the new subscriptions received since the last propagation, (ii) the subsumption probability, and (iii) the number of brokers whose summaries are being merged at the sending broker.

To measure the bandwidth requirements (figure 8) we measured the total bytes sent so that all brokers' subscriptions are fully propagated. In our approach the algorithm specified in section 4.2 is employed, batching σ subscriptions in every period and sending them, as prescribed by the algorithm. For Siena, we followed the description of their approach [6]: for every broker B a minimum spanning tree is formed and the subscriptions

are forwarded from neighbor to neighbor from B until they have reached all brokers or until they are subsumed.

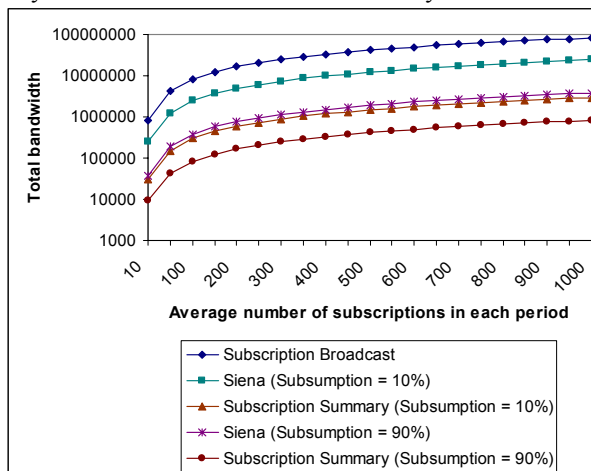


Figure 8: Bandwidth for subscription propagation.

In figure 8 the y-axis is logarithmic. We first state that obviously both our approach and Siena significantly outperform the baseline approach by orders of magnitude with respect to network bandwidth requirements. We further stress that (i) even for a value of σ (number of subscriptions in each period) that is a very small percentage of S (e.g., 1% of S) our approach is better than its competitors. Note that this is important since small values of σ imply small latencies before the subscription summaries are sent; (ii) scalability is very good, as evidenced by the nearly 'flat' lines; and (iii) that for large-scale systems with large numbers of active users submitting subscriptions (i.e., large σ values), our approach is up to three orders of magnitude better when compared to the baseline approach. When compared against Siena's subsumption approach we stress that we drastically outperform it (by a factor of 4 to 8 times).

Hop Count Performance

Figure 9 presents the number of hops for subscription propagation. Hop-counts reflect the number of brokers involved in the process of subscription propagation. We count as one hop every message that is being sent from a broker to another (regardless of whether the two brokers are neighbors in the overlay topology). With these hop counts we wish to count the number of brokers involved in the operation.

We note a significant difference in figure 9 between the two approaches. This difference is due to (i) the fact that our algorithm utilizes the merging of subscription summaries at each broker and (ii) that at each broker our algorithm forwards the merged subscription to only one neighbor. Hence, the propagation of the global knowledge about all brokers' subscriptions in our approach always requires a number of hops that is smaller than the number of brokers in the system. In Siena every broker propagates

received (and own subscriptions) to each neighbor with a probability that depends on the subsumption probability. In the worst case in Siena (subsumption percentage = 0%) each broker will receive the subscriptions of all others (for a total of 24 times 23 hops).

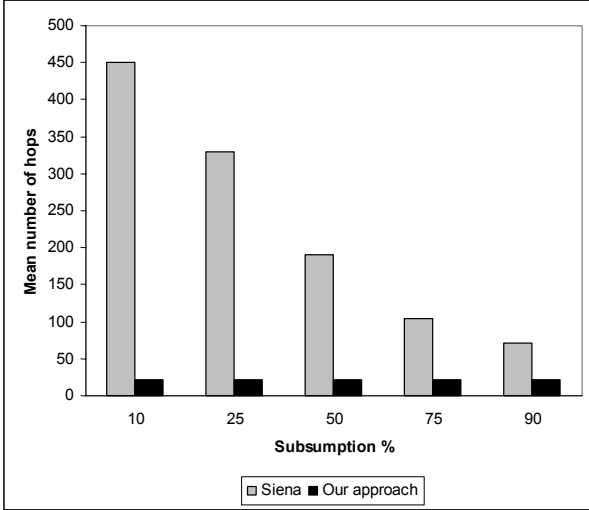


Figure 9: Mean hops needed for subscription propagation.

5.2.2. The performance of distributed event processing

We measure the number of hops needed for an event to be routed to all matched brokers. In our approach a broker knows exactly the broker to which the event will be forwarded. In Siena, the routing paths for events are set by subscriptions, which are propagated throughout the network from neighbor to neighbor in the overlay. When a producer publishes an event, matching the subscription, the event is routed following the reverse path put in place by the subscription's propagation.

We study both methods for varying event popularities, which captures the number of brokers that match the event; the 'matched' brokers are randomly chosen for every event. Figure 10 shows the mean number of hops required to process 24,000 events, 1000 events of each broker. Our algorithm is shown to be better for event popularities up to 75%. For very highly popular events, Siena is better due to the extremely high subsumption percentages it achieves. However, we expect that the majority of events will not be so popular.

5.2.3. Subscription Storage Requirements

Figure 11 shows the total storage requirements across all brokers, where each broker receives and propagates a varying number of subscriptions (from 10 to 1000) in each period. Our approach outperforms Siena by about two to five times for subsumption probability equal to 10% and 90%. The reason for this improvement lies in the

generalizations per attribute achieved by our per-broker summaries and extended by the multi-broker summaries and associated algorithms. Note that for small subsumption probabilities, Siena requires almost the same storage space as the baseline approach, as expected.

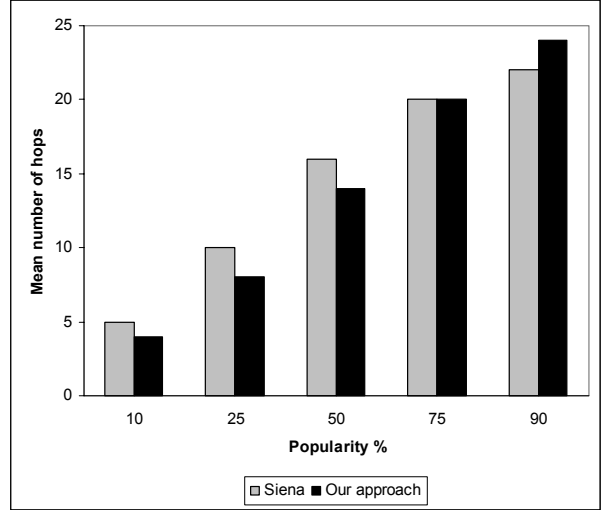


Figure 10: Mean hop counts in event propagation.

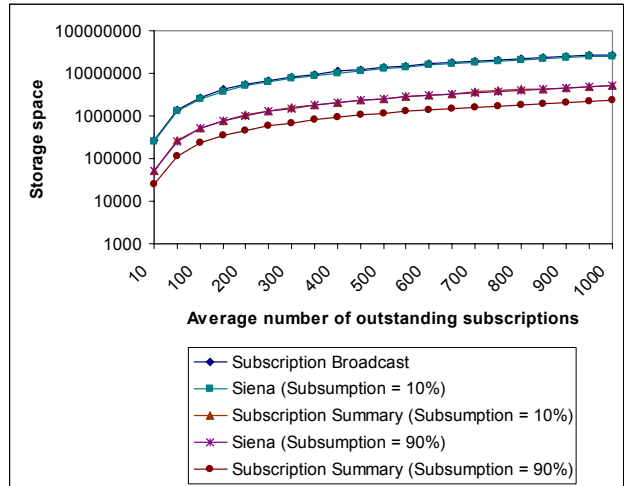


Figure 11: Storage requirements for subscriptions.

5.2.4. Computational demands for event processing

Briefly, the matching algorithm is dominated by its first two steps. The time needed for step 1 is equal to:

$$T_1 = n_{ae} \times \max\{n_{sr} \times L_a, n_e + L_a\} + n_{se} \times (n_r + L_s)$$

If P is the total number of subscriptions that have been collected from step 1, the computational time for step 2 is equal to: $T_2 = |P|$. Thus, the total computational time is $O(N)$, where N is the number of subscriptions. (For space reasons we omit the analysis for maintaining summaries, which is also $O(N)$). Despite the same complexity with related work, we expect that event filtering and matching

will be faster in our paradigm, given the summaries and the generalized attributes. Furthermore, we have shown that the number of involved brokers during event processing is, on average, smaller.

6. Conclusions

We contributed a new paradigm for pub/sub systems based on the novel notion of subscription summaries. We have presented the data structures constituting the summaries and the algorithms manipulating these summaries in order to match incoming events against the brokers' subscriptions. Also, we have developed the notion of multi-broker summaries, shown how to compute them, and the accompanying algorithm for subscription summary propagation. These contributions introduce significant performance gains during subscription propagation, for network bandwidth, storage requirements, and in required broker involvement (hop counts). We then contributed a distributed event processing algorithm, which ensures efficiency during the event processing phase, in terms of the number of brokers involved in event routing.

Our performance results show that our approach can drastically improve the bandwidth requirements to propagate subscriptions, outperforming the subsumption mechanism of Siena by a factor of four to eight. At the same time, the hop count for both subscription propagation and event processing is smaller and the computational requirements at each broker for filtering and matching events are expected to be better than those of related work. Finally, the storage requirements are smaller than Siena's by 2 to 5 times.

Our on-going work includes ensuring load balancing during event processing, which is an open problem. We employ 'virtual degrees' for the maximum-degree nodes, reducing their load, while continuing, however, to offer significant improvements. Further, we are currently extending our structures to accommodate dynamically-changing attribute schemata, for larger-scale networks (e.g., multi-ISP, global CDNs) (basically, this only requires changing the c_3 field of subscription ids). Finally, we are also currently developing techniques combining summarization and subsumption.

7. References

- [1] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching events in a content-based subscription system. *Proc. ACM PODC Symposium*, pp 53–61, 1999.
- [2] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajao, R. E. Strom, and D. C. Sturman. An efficient multicast protocol for content-based publish/subscribe systems. *Proc. ICDCS*, pp 262–272, 1999.
- [3] G. Banavar, M. Kaplan, K. Shaw, R. E. Strom, D. C. Sturman, and W. Tao. Information flow based event distribution middleware. *Proc. ICDCS Workshop on Electronic Commerce and Web Applications*, 1999.
- [4] Cable and Wireless plc. <http://www.cw.com>.
- [5] A. Carzaniga, E. Nitto, D. Rosenblum, and A. Wolf. Issues in supporting event-based architectural styles. *3rd Intl Software Architecture Workshop*, 1998.
- [6] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Achieving scalability and expressiveness in an Internet-scale event notification service. *Proc. ACM PODC*, pp 219–227, 2000.
- [7] G. Cugola, E. D. Nitto, and A. Fuggetta. The JEDI event-based infrastructure and its application to the development of OPSS WFMS. *IEEE TSE*, Sep. 2001.
- [8] F. Fabret, H. A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. *ACM SIGMOD 2001*, pp. 115-126, 2001.
- [9] J. Gough and G. Smith. Efficient recognition of events in a distributed system. *Proc. Australasian Computer Science Conference*, Feb. 1995.
- [10] R. E. Gruber, B. Krishnamurthy, and E. Panagos. The Architecture of the READY Event Notification Service. *ICDCS workshop*, June 1999.
- [11] G. Muhl, L. Fiege, F. Gartner, and A. Buchmann. Evaluating advanced routing algorithms for content-based publish/subscribe systems. *Proc. MASCOTS'02*, pp167-176, 2002.
- [12] Object Management Group. CORBA services – event service specification. Technical report 2001. <ftp://ftp.omg.org/pub/docs/formal/01-03-01.pdf>.
- [13] Object Management Group. CORBA services – notification service specification. Tech. report, 2000. <ftp://ftp.omg.org/pub/docs/formal/00-06-20.pdf>.
- [14] B. Oki, M. Pfluegl, A. Siegel, and D. Skeen. The Information Bus - an architecture for extensible distributed systems. *Operating Systems Review*, 27.5:58–68, 1993.
- [15] P. R. Pietzuch and J. Bacon. A distributed event-based middleware architecture. *Proc. 1st International Workshop on Distributed Event-Based Systems*, 2002.
- [16] A. Rowstron and P. Druschel. Pastry: Scalable decentralized object location and routing for large-scale peer-to-peer systems. *Proc. Middleware 01*, 2001.
- [17] B. Segall and D. Arnold. Elvin has left the building: A publish/subscribe notification service with quenching. *Proc. Australian UNIX Users Group Technical Conference*, pp 243–255, 1997.
- [18] SoftWired Inc. iBus. <http://www.softwired-inc.com>.
- [19] Sun Microsystems, Inc. Jini(TM) technology core platform spec - distributed events. Technical report, 2000. <http://www.sun.com/jini/specs>.
- [20] TIBCO Inc. TIB/Rendezvous. <http://www.tibco.com>.
- [21] P. Triantafillou, A. Economides. Subscription Summaries for Scalability and Efficiency in Publish/Subscribe Systems. *Proc. 1st International Workshop on Distributed Event-Based Systems*, 2002.
- [22] Vitria. BusinessWare. <http://www.vitria.com>