Efficient Matching for Content-based Publish/Subscribe Systems

Françoise Fabret
INRIA Rocquencourt
Francoise.Fabret@inria.fr

François Llirbat
INRIA Rocquencourt
Francois.Llirbat@inria.fr

João Pereira*
INRIA Rocquencourt
Joao.Pereira@inria.fr

Dennis Shasha
New York University
shasha@cs.nyu.edu

1 Introduction

It is widely accepted that the majority of human information will be on the Web in ten years. As pointed out in [6], besides systems for searching, querying and retrieving information from the Web, there is a need for systems being able to capture the dynamic aspect of the web information by notifying users of interesting events. This functionality is crucial for web users (or applications) who want to exploit highly dynamic web information such as stock markets updates or auctions. A tool that implements this functionality must be scalable and efficient. Indeed, it should manage millions of user demands for notifications (i.e. subscriptions); It should handle high rates of events (several millions per day) and notify the interested users in a short delay. In addition, it should provide a simple and expressive subscription interface and efficiently cope with high volatility of web user demands (new subscriptions, new users and cancellations). Finally, it should facilitate integration of similar kinds of information issued by different publishers (e.g. new auctions coming from distinct auction sites). We propose a system, called *Le Subscribe*, which addresses these issues.

To better illustrate these concepts let us consider the set of *auction sites* in the internet (e.g. *ebay*[5], *amazon*[2] or *yahoo*[15]). Every day, a large number of items is put up for auction in each of those auction sites. For example, *ebay* publishes about 560000 new auction items per day. Interested users need to access each site periodically, and repeat their queries, which may differ from site to site, to get the new interesting items.

The classical approach for query subscription is a mediator system where queries are periodically evaluated against static data. This static approach does not scale for high rate of events and a large number of volatile subscriptions, since it requires the storage of large event histories between two successive computations and requires repeated complex multi-query optimization. The publish/subscribe mechanism can also be applied in this context. This approach is different from the former one, since the events are processed on-the-fly to discover the matching subscriptions.

Publish/subscribe systems, shortly named *pub/sub* systems, establish a connection between publishers (producers) and subscribers (consumers) of events, behaving as a

*Founded by “Instituto Superior Técnico” - Technical University of Lisbon and by a JNICT fellowship of Program PRAXIS XXI (Portugal)

mediator between publishers and subscribers. This way, publishers are decoupled from subscribers, they do not need to be aware of each other. Publishers submit events to the pub/sub system which is responsible for notifying the interested subscribers. Subscribers specify the events they are interested in to the pub/sub system through a *subscription* language. Usually, pub/sub systems represent events in the form of a set of attribute-value pairs. In the auction example, each event (item) can be represented by attributes price (initial price of the item), category (auction sites classify their items according to the category they belong to, e.g. *Car*, *Book*, *Toy*) and description (describes the item in a summary way).

There are two kinds of pub/sub systems: *subject-based* and *content-based*. In subject-based systems, events are classified by groups or subjects and can be filtered only according to their group. A subject-based system would assign a group for each category. Publishers publish an event labeling it with the corresponding item's category as its group's name, and subscribers define just the groups they are interested in. Examples of such systems are iBus[10], TIB/Rendezvous[4] and OrbixTalk[11]. Content-based systems are an emerging type of pub/sub system where events are filtered according to their attribute values, using filtering criteria defined by the subscribers. This way, subscribers can specify, for example, they are interested in items of category *Toy* with a price lower than 5. Examples of content-based systems are Gryphon[3], NEONet[13], READY[8] and publish/subscribe mechanisms integrated in commercial DBMS products like Oracle8i, SQL Server 7.0, or Sybase.

Compared to subject-based systems, content-based systems offer more subscription expressiveness. The cost of this gain in expressiveness is an increase in the complexity of the matching process: the more sophisticated the constructs, the more complex the matching process. This complexity combined with a large number of subscriptions may severely degrade the matching efficiency. So, systems devoted to support a large number of subscriptions and a high rate of events have to face a tradeoff between the subscription language sophistication and matching efficiency. Le Subscribe is a content-based system where we designed an expressive language that lends itself to very efficient matching. Our main contributions in Le Subscribe are:

- A semi-structured event model which is well suited for the information published on the Web, and flexible enough to support easy integration of publishers.
- A subscription language which is designed to be simple while supporting the most usual queries on event notifications.
- An efficient matching algorithm for processing events in real time which can handle a large number of volatile subscriptions (several millions) and supports high event rates (several millions per day).
- Simple interfaces for publishing and subscribing which enable an easy integration of the system in the *Web*. The system supports both HTTP protocol and Java RMI.

This paper describes the event model, subscription language and the matching algorithms developed for our context-based pub/sub system. The event model and subscription language adopted in our pub/sub system are presented in Section 2. in Section 3 we describe some efficient matching algorithms that can be used in Le Subscribe. In this section we also define the matching problem in a formal way. Some experimental results are shown in Section 4. Finally our conclusions are presented in Section 5.

2 Mediation functionalities of the pub/sub system

As mentioned in the introduction, a pub/sub system behaves as a mediator between publishers and subscribers. One of the problems it has to solve is an integration problem.

Indeed, it has to mask the heterogeneity of similar information sent by different publishers in order to allow subscribers to specify their requirements without having to face with heterogeneous information. Publishers should publish their events according to an event schema defined in the system and subscribers define their interests in this event schema using a subscription language. Our solution to the pub/sub mediation problem lies in the definition of:

- An integrated event schema for modeling the events published via the pub/sub system,
- An integration model,
- And a subscription language over this schema.

In what follows we detail the different aspects of the solution.

2.1 Integrated event schema

The purpose of the integrated event schema (IE schema for short) is to provide the description of the conceptual scheme of the information (i.e. the events) published via the pub/sub system. An IE schema S is a sixtuple ($\mathcal{A}$, ET, $\mathcal{D}$, *dom*, $event\_type$, $\mathcal{E}$) where:

- $\mathcal{A}$ is a set of attributes, each of them denoted by an unique identifier: its name. Among the attributes contained in $\mathcal{A}$ there is a distinguished attribute named $event\_type$.
- Each attribute in $\mathcal{A}$ has a domain that may be *numeric*, *string*, *enumerated* or *hierarchical*. The hierarchical domain is an enumerated domain whose elements are organized according to a hierarchy and is useful to depict categories and subcategories. $\mathcal{D}$ represents the set of attribute domains. Given an attribute A of $\mathcal{A}$, its domain is computed by function *dom* $\mathcal{A} \longrightarrow \mathcal{D}$. Apart from the data type of the values, the description of a domain in the IE schema includes the specification of comparison operators over the values. For example, numeric and string domains are totally ordered, they implicitly support standard comparison operators, like $>$ or $<$, with a standard semantics. With a domain of an enumerated type, it is possible to associate a certain partial ordering. Such a domain will also support some comparison operators. Nevertheless their semantics are specific to the domain.
- ET represents the domain of the distinguished attribute $event\_type$. So ET is one of the domains in $\mathcal{D}$. This domain is of enumerated type, it consists in a set of identifiers.
- Each element of ET is associated with an *event schema*. Given e an element of ET, its event schema is a set of the form $\{(A_1, n_1, u_1)$, ..., $(A_p, n_p, u_p)\}$ where for $i = 1..p$ A_i denotes an attribute of $\mathcal{A}$, and n_i and u_i denote two annotations. The former ranges over values in $\{$mandatory, optional$\}$, the later over values in $\{$unique, multiple$\}$. The elements of an event schema are pairwise different over their attributes, i.e. each attribute of $\mathcal{A}$ appears at most once in the event schema of e. $\mathcal{E}$ is the function which computes the event schema of the elements of ET.

Suppose the following example: an object of type *antiques* is described by three mandatory attributes *price*, *period* and *quantity*. An object of type *furniture* can be described using also three mandatory attributes: Attributes *price* and *quantity* are in common with the *antiques* event type; Attribute *furniture_category* has a hierarchical domain ranging over furniture categories which can be organized in bedroom, dining room, outdoor categories and sub-categories like table, chair, ... Furniture description could be enriched with the optional attribute *material*. In this case, we have the following:

- $\mathcal{A} = \{price, period, quantity, furniture\_category, material, event\_types\}$.
- $ET = \{antiques, furniture\}$.
- $\mathcal{E}(furniture) = \{(price, mandatory, unique), (quantity, mandatory, unique), (furniture\_category, mandatory, unique), (material, optional, unique)\}$.

Definition (*Attribute set, Mandatory set, Unique set*) Let S be an IE schema, and e an element of ET. Then the *Attribute set* for e is the set of the attributes occurring in the event schema of e (i.e. in $\mathcal{E}(e)$) The *Mandatory set* (resp. the *Unique set*) for e is similar with the attribute set, excepted that only attributes having a mandatory notation (resp. an unique notation) in $\mathcal{E}(e)$ are retained. For example, the Mandatory set for *furniture* is constituted by attributes price, quantity and furniture_category.

View over an IE schema Let S = ($\mathcal{A}$, ET, $\mathcal{D}$, *dom*, $\mathcal{E}$) be an IE schema. Then a view $\mathcal{V}$ over S is a pair (E, R) where E is a set of event types occurring in ET, and R is a set of (A, n, u) triples satisfying the following properties:

1. for each $(A, mandatory, u)$ triple occurring in $\cup_{e \in E}\mathcal{E}(e)$, there is a triple $(A, mandatory, u')$ in R,
2. if a triple (A, n, u) occurs in R then A occurs in the attribute set of some e of E,
3. the (A, n, u) triples occurring in R are pairwise different over their attributes: mandatory and multiple annotations have priority over the optional and unique ones,

Event instance Let S = ($\mathcal{A}$, ET, $\mathcal{D}$, *dom*, $\mathcal{E}$) be an IE schema, and $\mathcal{V}$ = (E, R) a view over S. Then an event instance ei over S with respect to $\mathcal{V}$ is a collection of attribute, set of values pairs satisfying the following consistency rules:

R1 ei contains the pair $(event\_type, E)$

R2 for each element (A, n, u) of R, ei contains one pair (A, V) with $V \subseteq dom(A)$, and, if u has the *unique* value, then V is a singleton.

R3 Only pairs satisfying rule $R1$ or $R2$ occur in ei.

Suppose that a publisher produces information concerning furniture which is also an antique. In this case, the events published by this publisher should belong to both event types. Consider the following published events $e_1 = \{(event\_type, (antiques, furniture))$ $, (price, 200), (quantity, 2), (period, XVI), (furniture\_category, table)\}$ and $e_2 = \{(event\_type, (antiques, furniture)), (price, 100), (quantity, 1), (period, XVIII)\}$. Event e_1 satisfies all the consistency rules while e_2 does not satisfy rule R2 (and obviously R3), since the mandatory attribute furniture_category is missing in e_2.

2.2 Integration model

IE schema provides an integrated representation of the publications issued by the publishers. In this section we present the integration model used in our pub/sub system. By integration model we mean the description of:

- the publication language over the IE schema: its syntax, its semantics
- the way the IE schema is built from the informations to publish.

Publication language In our system, the only way to issue a publication is to express it in the form of an *event instance* over the IE schema. The semantics of an event instance are as follows. Let ei be an event instance in its canonical form. The pair having event_type as attribute essentially serves to check the consistency rules. Apart from this specific pair, a pair (A, V) means that an instantiation of A by any value in V is valid in ei.

IE schema modification rules We propose a very simple integration schema in which the IE schema may be seen as the union of the publishers schema (by schema of a given publisher we mean the part of the IE schema which is used by this publisher to issue its event instances). This is achieved by allowing a publisher to extend the IE schema. There are four schema extension rules. Let $S = (\mathcal{A}, ET, \mathcal{D}, dom, event\_type, \mathcal{E})$ be an IE schema

Domain creation rule adds a new domain in $\mathcal{D}$ and specifies the comparison operators over this domain.

Attribute creation rule adds a new attribute in $\mathcal{A}$, and extends the dom function to this attribute.

Event schema extension rule extends the event schema of some event type e by adding a triple $(A, optional, u)$ to $\mathcal{E}(e)$ where A is an attribute of $\mathcal{A}$ which did not occur in the attribute set of e.

Event schema creation rule extends ET by adding a new event type, and also extends the $\mathcal{E}$ function to this new event type (i.e. specify its event schema).

The effect of a schema modification is global. So every time a publisher extends the integrated schema, the extended schema is made available for all publishers. Applying these rules cannot generate integration conflicts between publishers or IE schema inconsistencies. Nevertheless schema extensions may generate some redundancies in the schema; for example, it may be that applying the domain creation rule leads a publisher to create a new domain having exactly the same definition than an existing one. We do not consider this problem, and do not propose any consolidation mechanisms to preserve IE schema minimality.

2.3 Subscription language

A *subscription* is defined as a conjunction of elementary predicates. The language provides predicates of the form $X\theta y$, where X is an attribute name, y is a value belonging to the domain of X and θ is a comparison operator. In the case of hierarchical domains $\leq$ and $<$ operators are semantically equivalent to the standard *is a kind of* relationship. In addition, the language supports X *contains* y predicates. Such predicate is true for an event instance e, if value y occurs in the set of values associated with attribute X in e. For example [*(event_type contains antiques) and (furniture_category* $<$ *dining table)*] describes a subscription for all events concerning any antique which belongs to any sub-category of dining tables.

An event instance e matches a subscription s if e provides a binding for every attribute occurring in s and all predicates of s are true with respect to this binding. A subscription is satisfied by any matching event instance.

3 Matching algorithms

The matching problem can be formulated as the following question: Given an event e and a set S of subscriptions which are the subscriptions of S satisfied by e? In this section we present several algorithms, called *matching algorithms* that intend to solve this problem. The main goal of any matching algorithm is to compute the set of subscriptions matched by one event or a set of events. In order to achieve that, we consider that the matching algorithm handles the events to match one at a time. Among the several matching algorithms described in this section, some of them were developed by us and the others were found in the literature. We present a comparative analysis of both.

3.1 Reformulation of the matching problem

For any matching algorithm there is always a pre-processing which is responsible for translating the set of subscriptions to match to an internal representation. This internal representation is used by the algorithm to determine the subscriptions matched by an event. In this section we will not describe the pre-processing associated to each algorithm, and present only the internal representations used in each one.

A simple algorithm for solving the matching problem consists of testing all subscriptions, one by one, against each incoming event. This *naive* algorithm is represented in figure 1. The *pred_match* function invoked in line 6 checks a predicate against the given event returning true if the predicate is verified and false otherwise. In the worst case, the time complexity of this algorithm in order to process an event e is $O(\sum_{s \in S}(\#($ predicates of s $)C_p)$, where C_p represents the cost of matching one predicate. In general, the performance of the naive algorithm degrades as the number of subscriptions increases.

Usually, a matching algorithm may be used in environments with a large number of subscriptions (several hundreds of thousands or even more). In such environments, if the response time to process an event is an important factor or the rate of events to process is high, the naive algorithm is not a satisfactory solution. The main problem with this algorithm is the existence of a high redundancy in the evaluation of the predicates. In fact, the same predicate can be evaluated as many times as the number of times it appears in the set of subscriptions.

The idea generally developed to cope with this drawback is to focus on predicates instead of subscriptions. The matching algorithm must then avoid the reevaluation of predicates by factorizing the subscriptions over their predicates. It may also use some deduction to decrease even more the number of predicates evaluated. For example, if predicate $(price, \leq, 10)$ is verified by an event then the predicate $(price, \leq, 20)$ is also verified; but if predicate $(price, =, 10)$ is verified, predicate $(price, =, 20)$ cannot be verified. With such a solution, the set of predicates that hold for a given event e can be computed in a very efficient way. The matching problem can now be reformulated as: knowing the predicates that are evaluated by an event which subscriptions are satisfied?

```
naive_match($S$, $e$)
1  // $S$ is the set of subscriptions, $e$ is the event to match
2  $matched \leftarrow \{\}$
3  foreach $s \in S$ do
4    $m \leftarrow$ true
5    foreach $p \in s.preds$ do
6      if $\neg\ predmatch(p, e)$ then
7        $m \leftarrow$ false
8        break // leave loop for
9      endif
10   endloop
11   if $m$ then
12     $matched \leftarrow matched \cup s$
13   endif
14 endloop
```

Figure 1: Naive algorithm.

```
fair_pred_match ($e$)
1  // $e$ is the event to process
2  $matched \leftarrow \{\}$
3  // Step 1 - compute the predicates satisfied by $e$
4  $satisfied\_preds \leftarrow eqpredmatch(e)$
5  $satisfied\_preds \leftarrow satisfied\_preds \cup lesspredmatch(e)$
6  $satisfied\_preds \leftarrow satisfied\_preds \cup greaterpredmatch(e)$
7  // Step 2 - counts the number of satisfied predicates for each subscription
8  foreach $p \in satisfied\_preds$ do
9    foreach $s \in pred\_to\_subs[p]$ do
10     $hitcount[s] \leftarrow hitcount[s] + 1$
11   endloop
12 endloop
13 foreach $s \in S$ do
14   if $hitcount[s] = \#subscriptions[i]preds$ then
15     $matched \leftarrow matched \cup s$
16   endif
17 endloop
```

Figure 2: Fair predicate algorithm.

In what follows, we are going to present two algorithms that answer the new formulation of the matching problem.

Fair predicate approach The first algorithm (figure 2), designated as *fair predicate algorithm*, was developed by us. During the pre-processing of this algorithm, the predicates are clustered by comparison operator and attribute. An association table, *pred_to_subs*, is maintained to make the correspondence between each predicate and the subscriptions it appears in. Each predicate is stored just once.

The *fair-predicate* algorithm is a two-step algorithm and consists basically of the following idea. The first step (lines 4-6) computes the satisfied predicates by applying the event to the set of all predicates specified in subscriptions. The *eqpredmatch*, *lesspredmatch* and *greaterpredmatch* functions compute the *equality*, *less than* and *greater than* predicates satisfied by a given event e, respectively. The second step (lines 8-17) computes the set of satisfied subscriptions from the set of satisfied predicates found in the previous step. The number of satisfied predicates by subscription is counted using an association table (lines 9-12). Finally, we compare the number of satisfied predicates with the number of predicates specified for each subscription (lines 13-17). A subscription is matched if both numbers are equal.

The *eqpredmatch* function searches, for each event's attribute, the predicate with the same value as the attribute value in the *equality* cluster associated to the attribute. If each cluster of equality predicates is kept suitably ordered, the worst-case time complexity to compute the satisfied equality predicates is $O(\sum_{a_i \in e.atts}(\log(\#D_i)+1))$ if a binary search algorithm is used for each cluster of equality predicates and $O(\#e.atts)$ if a hash table is used. $e.atts$ represents the attributes of the event.

The *lesspredmatch* searches, for each event's attribute, the predicate with the greatest value less than or equal to the attribute value in the *less than* cluster associated to the attribute. After having found this predicate, we know that all predicates that are placed before (assuming the predicates are stored in increasing order) are also satisfied by the event. We can use a binary search algorithm in this case as well to find the predicate with the greatest value less than or equal to the attribute's value. The worst-case time complexity to compute the satisfied predicates in this case is $O(\sum_{a_i \in e.atts}(\log(\#D_i) + \#D_i))$. The *greaterpredmatch* function is similar to *lesspredmatch*, having the same worst-case time complexity.

As shown above, this algorithm evaluates each predicate at most once. Its main drawback is the processing necessary to compute the matched subscriptions from the satisfied predicates. There may be a large number of sums to do and the number of comparisons is equal to the number of subscriptions. The time complexity of this algorithm is $O(C_= + C_\leq + C_\geq + C_{add}P_{sat} + C_{comp}(\#S))$, where $C_=$, $C_\leq$ and $C_\geq$ represent, respectively, the cost of computing the satisfied *equality*, *less than* and *greater than* predicates. C_{add} represents the cost of an addition and P_{sat} is the number of satisfied predicates by the event. C_{comp} represents the cost of a comparison and S is the set of subscriptions to match.

Gough algorithm Gough and Smith propose a matching algorithm in [7] which we call the *gough* algorithm (see figure 3). Here, the subscriptions are translated into a tree. This tree is organized in such a way that if an event matches one or more subscriptions, there is only a single path to follow in order to find out the matched subscriptions. Each path factorizes the subscriptions that can be matched by the attribute values corresponding to the path followed. A subscription can correspond to several paths in the tree. The leaf nodes store the matched subscriptions. The number of levels of the tree is equal to the number of attributes. At each level of the tree (lines 5-13), a certain event attribute value is checked in order to determine the next node to follow (lines 8-13) until a leaf-node is reached (lines 5-6). The next node is chosen if there exist an edge between the current node and another node whose associated value is equal to the attribute value. If, at a given level, there is not such an edge (line 10), it implies that the event does not match any subscription.

Each predicate is evaluated at most once as in the *fair-predicate* algorithm, but now the matched subscriptions are found in a faster way. If the values stored in each node are conveniently ordered, a binary search algorithm can be used at each level to find out the next node triggered by the event. The worst-case time complexity of this algorithm is $O(\#attributes \log(\#subscriptions))$ so it is sublinear with the number of subscriptions. Nevertheless, there is a redundancy in the way predicates are stored since a predicate can be stored several times. Moreover, in the worst-case there is a combinatorial explosion of the number of times a predicate has to be stored.

By analyzing the *fair predicate* and *gough* algorithms, we conclude that they both optimize the number of times each predicate is evaluated. Each predicate is evaluated at most once for each event to process. Nevertheless, the algorithms take two opposite approaches. The fundamental difference between them is that *fair predicate* algorithm

```
 gough_match($e$)
1  // $e$ is the event to process. $T$ holds the subscription tree.
2  $matched \leftarrow process$(T.root,e)
3
4process($n$, $e$)
5  **if** $n$ is a leaf node **then**
6    **return** $n.subscriptions$
7  **endif**
8  $next \leftarrow$ next node triggered by $e$ at node $n$
9  **if** $next$ is NULL **then**
10   **return** $\{\}$
11 **else**
12   **return** $process(next, e)$
13 **endif**
```

Figure 3: Gough algorithm.

optimizes the space required to store the predicates while the *gough* algorithm optimizes the execution time of the algorithm.

The *gough* algorithm is faster because it stores the predicates in such a redundant way that it is able to figure out in a very efficient way the matched subscriptions. Its drawback is that it has a combinatorial explosion in the number of times each predicate is stored. The *fair predicate* algorithm is slower than the *gough* algorithm to compute the matched subscriptions but it stores the predicates in an efficient way since each predicate is stored only once.

Another important factor, besides time and space, that should be taken into account to characterize a matching algorithm is its maintainability. This factor corresponds to the time needed to update the data structures used by the matching algorithm when the set of subscriptions is modified. The importance of the maintainability in the performance of the matching algorithm depends on the rate of modifications applied to the set of subscriptions. If a modification is very rare, the maintainability factor may be neglected. But, if the rate of modifications is of the same order as the rate of the events, this factor is very important. The maintainability depends essentially on the amount of redundant data that is used by the matching algorithm. The more redundant data the matching algorithm uses the more time is spent to update the data structures used.

The *gough* algorithm is not easily maintainable. [7] suggests that new modifications to the set of subscriptions are made by constructing the subscription tree from scratch. The *fair predicate* algorithm is easily maintainable. Subscriptions can be added or removed to the internal data structures used by this algorithm in an incremental way. For example, adding a new subscription corresponds to updating the data structures that hold the predicates if a new predicate is specified in the subscription and to add a new entry in the association table that establishes the correspondence between the new subscription and its predicates.

Other solutions to the matching problem will be placed between the three algorithms mentioned above: *naive*, *fair predicate* and *gough*. They have a lower redundancy predicate evaluation than the *naive* algorithm but higher than the *fair predicate* and *gough* algorithms. In the space dimension, the other algorithms have a higher redundancy in what concerns predicate storage than the *fair predicate* algorithm but lower than the *gough* algorithm. As it is shown by the *gough* and *fair predicate* algorithms there is a trade-off between space and time dimensions. In order to be faster than the *fair predicate* algorithm, we need to define new data structures that factorize the subscriptions using more than one predicate as does the *gough* algorithm. This way, the computation of the matched subscriptions can be made in a more efficient way. Nevertheless, this

```
 eq_pref_match ($e$)
1  $matched \leftarrow \{\}$
2  // Step 1: compute the satisfied equality subscriptions and nonequality predicates
3  $partial\_satisfied\_sub \leftarrow eqsubmatch(e)$
4  $satisfied\_preds \leftarrow lesspredmatch(e)$
5  $satisfied\_preds \leftarrow satisfied\_preds \cup greaterpredmatch(e)$
6  // Step 2: counts the number of satisfied nonequality predicates for each subscription
7  foreach $p \in satisfied\_preds$ do
8    foreach $s \in pred\_to\_subs[p]$ do
9      $hitcount[s] \leftarrow hitcount[s] + 1$
10   endloop
11 endloop
12 foreach $s \in partial\_satisfied\_sub \cup neqsub$ do
13   if $hitcount[s] = subscriptions[s].n\_neqpreds$ then
14     $matched \leftarrow matched \cup s$
15   endif
16 endloop
```

Figure 4: Equality-preferred algorithm.

has a cost. First, there is a redundancy in the space dimension since a predicate can be stored several times. Second, there is a precomputation cost due to creating and maintaining the redundancy space.

We will now describe five more of those matching solutions and compare them with the *fair predicate* and *gough* algorithms. The algorithms: *equality-preferred* and *equality-preferred approach with nonequality quarantining* were developed by us. The others were presented in references: [9], [1] and [14].

Equality-preferred approach This algorithm, represented in figure 4, tries to solve the main drawback of the *fair predicate* algorithm. In this algorithm, subscriptions are *clustered* by their equality predicates. Two subscriptions with equality predicates over the same attributes are placed in the same cluster. Each cluster is associated to the set of attributes that appear in the *equality* predicates of the subscriptions considered, the cluster *schema*, and stores all combinations of equality predicates over that set of attributes. A subscription is placed at most in one cluster. The *nonequality* predicates are stored in the same way as in the *fair predicate* algorithm. The *equality-preferred* algorithm has to keep track of the subscriptions that do not have any equality predicate, which are stored in a data structure called *neqsub*.

This algorithm works in two steps as follows. In the first step, the subscriptions whose equality predicates are satisfied as well the satisfied nonequality predicates are computed . The *equality* clusters are used to compute the subscriptions whose equality predicates are satisfied by an event (line 3). The *nonequality* predicates are computed in a analogous way to the *fair predicate* algorithm (lines 4-5). In the second, step we count the number of *nonequality* predicates satisfied per subscription (lines 7-11). Finally, the subscriptions matched by the event are those having their *equality* predicates satisfied or having no *equality* predicate (which are stored in *neqsub*) and whose number of *nonequality* predicates is equal to the number of *nonequality* predicates satisfied by the event (lines 12-16).

The *eqsubmatch* function computes the subscriptions whose equality predicates are matched by the event. This function has to determine the equality clusters whose schema is contained in the *schema* of the event to process. For each cluster in such conditions, the event values corresponding to the cluster schema must be searched in the cluster in order to locate the partially matched subscriptions. If each equality cluster is kept suitably ordered, the worst-case time complexity to compute the partially matched subscriptions is $O(2^n \sum_n \log S_n)$ if a binary search algorithm is used for each cluster or $O(2^n)$ if a hash table is used, where n is the number of attributes and S_n is the size of cluster n. The time complexity of this algorithm is $O(C_= + C_\leq + C_\geq + C_{add}(\#P_{neq\ sat}) + C_{comp}(\#S_{partial\ sat} + \#S_{neq}))$, where $C_=$, $C_\leq$ and $C_\geq$ represent, respectively, the cost of computing the subscriptions whose equality predicates are satisfied and the satisfied *less than* and *greater than* predicates. C_{add} represents the cost of an addition and $P_{neq\ sat}$ is the number of *nonequality* predicates satisfied by the event. C_{comp} represents the cost of a comparison and $S_{partial\ sat}$ and S_{neq} are, respectively, the the set of partially satisfied subscriptions and of *nonequality* subscriptions.

The matched subscriptions are computed in a similar way as in the *fair predicate* algorithm but doing a smaller number of sums and comparisons since the matched *equality* predicates are not taken into account. There is a redundancy in this algorithm as the same equality predicate can be stored several times in different clusters and can also be evaluated several times (but no more than the number of existing clusters). The maintainability cost, in this case, is higher than for the *fair-predicate* algorithm (but much smaller than for the *gough* algorithm) as it becomes necessary to maintain the equality clusters too.

Equality-preferred approach with nonequality quarantining This algorithm, shown in figure 5, takes a different approach from the *fair predicate* and *equality-preferred* algorithms. It keeps track of the subscriptions without *equality* predicates, *neqsub* in the figure. We call these subscriptions the *nonequality* subscriptions. Only the predicates belonging to *nonequality* subscriptions are clustered by attribute and comparison operator instead of all *nonequality* predicates like in the *equality-preferred* algorithm. An association table, *pred_to_neqsubs*, between the *nonequality* predicates and the corresponding *nonequality* subscriptions is maintained. As in the previous algorithm, the subscriptions are clustered by their *equality* predicates. We will shortly refer to this algorithm as *nonequality quarantining* in the rest of this paper.

```
   quarantining_match ($e$)
1  $matched \leftarrow \{\}$
2  // Step 1: processes the subscriptions with equality predicates
3  $partial\_satisfied\_sub \leftarrow eqsubmatch(e)$
4  foreach $s \in partial\_satisfied\_sub$ do
5    if $match(s, e)$ then
6      $matched \leftarrow matched \cup s$
7    endif
8  endloop // Step 2: computes the nonequality predicates satisfied by $e$
9  $satisfied\_preds \leftarrow neq\_lesspredmatch(e)$
10 $satisfied\_preds \leftarrow satisfied\_preds \cup neq\_greaterpredmatch(e)$
11 // Step 3: determines the nonequality subscriptions matched by $e$
12 foreach $p \in satisfied\_preds$ do
13   foreach $s \in pred\_to\_neqsubs[p]$ do
14     $hitcount[s] \leftarrow hitcount[s] + 1$
15   endloop
16 endloop
17 foreach $s \in neqsub$ do
18   if $hitcount[s] = subscriptions[s].n\_preds$ then
19     $matched \leftarrow matched \cup s$
20   endif
21 endloop
```

Figure 5: Equality-preferred approach with nonequality quarantining algorithm.

This algorithm is a three step algorithm. Step one finds the subscriptions with equality predicates which are satisfied by the event. First, the subscriptions whose *equality* predicates are satisfied are computed (line 3). This is done in the same way as in the *equality-preferred* algorithm (the function *eqsubmatch* is equal to the one used in the *equality-preferred* algorithm). Then, (lines 4-7) verifies for each selected subscription if its *nonequality* predicates (if exist) are all satisfied by event e (function *match*). If the *nonequality* predicates are satisfied or if the subscription have none, the subscription is matched by e. The other two steps of this algorithm are similar to the *equality-preferred* algorithm. The *nonequality* predicates of the *nonequality* subscriptions which are satisfied by the event are determined in lines 9 and 10. The functions *neq_lesspredmatch* and *neq_greaterpredmatch* are similar to *lesspredmatch* and *greaterpredmatch*, respectively, but they consider just the *nonequality* predicates of the subscriptions without *equality* predicates instead of considering all subscriptions. In the last step, the number of satisfied predicates for each *nonequality* subscription is counted (lines 12-16) and then this number is compared with the number of predicates of the subscription (lines 17-21). If both numbers are equal then the *nonequality* subscription is matched.

Compared to the *equality-preferred* algorithm , the *nonequality quarantining* algorithm may evaluate the *nonequality* predicates more than once since it verifies, for each partially satisfied subscription, if its *nonequality* predicates are all satisfied. Nevertheless, the number of sums and comparisons is drastically reduced if we consider that the number of *nonequality* subscriptions is much smaller than the total number of subscriptions. The *nonequality* predicates of the subscriptions with equality predicates are stored together which each subscription. Therefore, the *nonequality* predicates may also be stored several times.

The time complexity of this algorithm is $O(C_= + C_\leq + E_\geq + C_{mp}(\#S_{partial\ sat}) + (\#S_{neq\ sub})C_{comp} + (\#P_{neq\ sat})C_{add})$, where $C_=$, $C_\leq$ and $C_\geq$ represent, respectively, the cost of determining the partially satisfied subscriptions, the satisfied *less than* and *greater than* predicates of the *nonequality* subscriptions. $P_{neq\ sat}$ represents the set of such satisfied predicates. C_{mp}, C_{comp} and C_{add} represent, respectively, the cost of verifying the *nonequality* predicates of a subscription, of doing a comparison and a sum. $S_{partial\ sat}$ and $S_{neq\ sub}$ represent the set of partially satisfied subscriptions and of *nonequality* subscriptions, respectively.

Hanson algorithm Hanson et al propose another matching algorithm in [9] which we call the *hanson* algorithm (see figure 6). During the pre-processing phase, this algorithm chooses the most selective predicate and places it in a *interval binary search* tree (*ibs*) associated to the predicate's attribute. An *ibs* tree is a one-dimensional index which allows efficient searching to determine which equality and nonequality predicates are satisfied by a value. The other predicates of each subscription are stored in a table, called *predicates*.

This algorithm is an improvement of the *naive* algorithm. It has two phases. In the first one (lines 4-7), it computes the set of subscriptions whose most selective predicate is verified by the given event. For each event's attribute value it searches the subscriptions partially matched by the value in the corresponding *ibs* tree. In the second phase (lines 9-20), the naive algorithm is applied to this set of selected subscriptions to determine the matched subscriptions. The other predicates of each selected subscription are evaluated one at a time; if all of them are verified, the subscription is matched by the event.

Each predicate may be stored several times as this algorithm factorizes only one predicate per subscription. A predicate can also be evaluated several times since each of the less selective predicates of each partially matched subscription have to be evalu-

```
hanson_match($e$)
1  // $e$ is the event to match
2  $matched \leftarrow \{\}$
3  // Step 1 - computes partially satisfied subscriptions
4  $partial\_matched \leftarrow \{\}$
5  **foreach** $(a_i, v_i) \in e.atts$ **do**
6    $partial\_matched \leftarrow partial\_matched \cup searchfor(ibs[a_i], v_i)$
7  **endloop**
8  // Step 2 - verifies other predicates
9  **foreach** $s \in partial\_matched$ **do**
10   $m \leftarrow$ **true**
11   **foreach** $p \in predicates[s]$ **do**
12     **if** $\neg\, predmatch(p, e)$ **then**
13       $m \leftarrow$ **false**
14       **break** // leave loop for
15     **endif**
16   **endloop**
17   **if** $m$ **then**
18     $matched \leftarrow matched \cup s$
19   **endif**
20 **endloop**
```

Figure 6: Hanson algorithm.

ated. In order to have a satisfactory performance, this algorithm requires a knowledge about the selectivity of each predicate. If this knowledge is not available or the predicates are not very selective, the number of subscriptions to check in the second phase of the algorithm may be very large and the performance turns out to be poor. The complexity time of this algorithm is $O(n_{att}C_{ibs} + n_{ms}(\bar{n}_p - 1)C_p)$. C_{ibs} represents the cost of searching an *ibs* tree which requires time $O(\log(N) + L)$ if the *ibs* tree is balanced, where N is the number of predicates indexed in the *ibs* tree and L is the number of predicates satisfied by a given event attribute value. C_p is the cost of evaluating a predicate of a subscription. n_{att}, n_{ms} and $\bar{n}_p$ represent the number of attributes, of subscriptions partially matched by an event and the average number of predicates per subscription, respectively.

The maintainability of this algorithm depends principally on the maintainability of an *ibs* tree. Each *ibs* tree should be kept balanced, otherwise the time of the worst-case to search the tree increases. The total average time to insert or delete a predicate from the *ibs* tree, and keep it balanced, is $O(\log^2 N)$.

Aguilera algorithm Aguilera et al propose a matching algorithm in [1] which we call the *aguilera* algorithm. This algorithm is similar to the *gough* algorithm. A subscription tree is also built but we may have to follow several paths while in *gough* algorithm there is at most one path to follow from the root to the leaf nodes to determine the matched subscriptions. Consequently, this algorithm has a higher time complexity but a much lower space complexity. Each predicate is seen as one possible result of applying a simple operation over an event attribute. Simple operations include getting the value of an event attribute or comparing an event attribute with a given value to verify it is higher or smaller. The first case corresponds to an equality predicate while the second corresponds to a *less than* or *greater than* predicate. The non-leaf nodes of the subscription tree represent a simple operation on an event attribute. Edges from a non-leaf node represent results of the associated operation and correspond to predicates present in the subscriptions to match. A leaf node n identifies the subscription that is described by the path followed from the root node to n. There is a special *don't care* edge, designed as

```
 aguilera_match($e$)
1  // $e$ is the event to process. $T$ holds the subscription tree.
2  $matched \leftarrow process$(T.root,e)
3
4process($n$, $e$)
5  $matched \leftarrow \{\}$
6  **if** $n$ is a leaf node **then**
7     **return** $n.subscriptions$
8  **endif**
9  $result \leftarrow$ apply $n.operation$ to $e$
10 **if** $\exists edge \in n.edgesedge.res = result$ **then**
11    $matched \leftarrow process(edge.node, e)$
12 **endif**
13 **if** $\exists * -edge \in n.edges$ **then**
14    $matched \leftarrow matched \cup process(* - edge.node, e)$
15 **endif**
16 **return** $matched$
```

Figure 7: Aguilera algorithm.

**-edge*, which represents the subscriptions that can be reachable through the edge and do not care about the result of the operation associated to the corresponding node.

Figure 7 describes in detail this matching algorithm. The subscription tree is walked down from the root node until the leaf nodes by selecting certain edges, at each node, according to the event attribute values. At each non-leaf node (lines 9-15), its associated operation is applied to the corresponding event attribute and the edge that represents the result of the operation (lines 10-12) as well as the **-edge* (lines 13-15) are followed if they are present. The set of subscriptions matched by an event is represented by the leaf nodes reached (lines 6-8) during the processing of the event to match. An inconvenient of this algorithm is that the fact that the number of paths to follow is equal to the number of subscriptions matched by the event. Other paths may have been followed partially due to the existence of **-edges* in the subscription tree. Thus is redundancy in predicate evaluation and storage due to the existence of the **-edges*.

For the case when there is only equality predicates, the number of levels of the subscription tree is equal to the number of event attributes and each level is assigned to a specific event attribute. The space required for the subscription tree is $O(NK)$, where N and K are, respectively, the number of subscriptions and attributes. The worst-case complexity time is $O(N(K+1))$. Nevertheless, it is shown in [1] that the *expected* time to match a random event is bounded by $(VK_1(K_1|S|^{1-\lambda}-1)(lnV+lnK_1))/((VK_1-1)lnK_1$, where $\lambda = lnV/(lnV + lnK_1)$, and $K_1 = K + 1$, and V is the number of possible values for each attribute. It is assumed the use of a hash table in each non-leaf node to search for the edge with the same value as the corresponding event attribute.

If *nonequality* predicates can appear in a subscription, then this algorithm is not so efficient. While a node may refer to several *equality* predicates, being each one represented by a different edge leaving the node, it can just refer to one *nonequality* predicate. Therefore, *nonequality* predicates increase the number of nodes (and levels) of the subscription tree and thus the number of nodes that must be processed in order to determine the matched subscriptions. An optimization is considered during the pre-processing phase of the algorithm in order to reduce the average number of nodes that need to be processed. Nevertheless, this optimization depends on the order the subscriptions are processed (placed in the tree) and the optimal node disposition can not be assured.

Relative to maintainability, subscriptions can be added or removed incrementally to the subscription tree. But, due to the way *nonequality* predicates are placed in the subscription tree, the number of nodes that need to be processed in order to place the new subscription in the tree may be much bigger than the number of predicates of the subscription to add. An algorithm for the addition case is defined in [1].

Neon algorithm The NEONRules commercial product [14] takes a different approach to solve the matching problem. In this algorithm there is not a factorization of predicates so predicates are stored as many times as they appear in the subscriptions. The existence of different event types is considered by this algorithm. Each event type corresponds to a different event schema. Subscriptions are associated to a certain event type and they are only evaluated if their event type corresponds to the event type of the event to process[1]. Each subscription is identified by a unique identifier. In the pre-processing phase of this algorithm the subscriptions are organized in the following way. There is a *predicates* table where the predicates of each subscription are stored. Each predicate stored in this table is described by the identifier of its subscription, the event type and event attribute it refers to, the comparison operator presented in the predicate and the referred constant value. The *operations* table is used to identify the existence of predicates with common characteristics, namely, refer to the same event type and attribute, and have the same comparison operator. This data structure allows the computation of the predicates in the *predicates* table that refer to the event to process ignoring all the others. The *n_predicates* table maintains the number of predicates for each subscription.

```
   neon_match($e$)
1  // $e$ is the event to process. $operations$ and $predicates$ are data
2  // structures filled in during the processing of the subscriptions.
3  $matched \leftarrow \{\}$
4  foreach $i \in operations$ such as $i.event\_type = e.type$ do
5     $value \leftarrow$ value of $e$ corresponding to attribute $i.attribute$
6     if $i.comparison = “EQ”$ then
7        foreach $pred \in predicates$ such as $pred.event\_type = i.event\_type \wedge$
8                 $pred.attribute = i.attribute \wedge pred.comparison = i.comparison \wedge$
9                 $pred.value = value$ do
10          if $\exists_{j \in and} pred.sub\_id = j.sub\_id$ then
11             $j.count \leftarrow j.count + 1$
12          else
13             add $(pred.sub\_id, 1)$ to $and$
14          endif
15       endloop
16       ... // similar to less and greater than comparison
17    endif
18 endloop
19 foreach $sub \in and$ do
20    if $sub.count = n\_predicates[sub.sub\_id]$ then
21       $matched \leftarrow matched \cup sub.sub\_id$
22    endif
23 endloop
```

Figure 8: The Neon algorithm.

This algorithm is shown in figure 8. It begins by finding all elements presented in the *operations* table which represent predicates that can be applied to the published event (line 4), i.e. have the field *event_type* equal to the event type of the published event. For each of such elements the event value corresponding to the event attribute identified by the element is obtained (line 5) and the predicates represented by this element are pro-

[1]In the other algorithms, the handling of several event types may be supported by defining the internal data structures used in the matching algorithm for each event type.

cessed to compute the satisfied predicates. This processing depends on the comparison operator used in those predicates. For the equality comparison (lines 6-15), the satisfied predicates correspond to those elements of the *predicates* table whose *event_type*, *attribute* and *comparison* fields are equal to the corresponding fields of the *predicates* element and whose *value* field is also equal to the previously obtained event value. For each located predicate that is matched by the event (lines 10-14), the corresponding element in the *and* data structure is updated. The other types of comparison operators are handled in a similar way to the equality comparison. Finally, after all *operations* elements have been processed, the matched subscriptions are determined (lines 19-23). A subscription is matched if its number of satisfied predicates is equal to its number of predicates.

In the solution presented by [14]an index over the fields *event_type*, *attribute_id* and *comparison* of the *predicates* table is defined in order to efficiently locate the predicates associated to each element of the *operations* table. Nevertheless, each located predicate must be evaluated afterwards. This number of located predicates can be large if a large number of subscriptions is defined for a certain event type. An index over the field *event_type* of the *operations* is also defined.

In what concerns space and time redundancy, since the predicates are not factorized in this algorithm, the same predicate may be stored and evaluated several times. An advantage of this algorithm is that it can be easily extended[2] to also allow predicates as *(att_1, comparison operator, att_2)*, where att_1 and att_2 refer to different attributes of an event. The maintainability of this algorithm corresponds to updating the tables *operations*, *predicates* and *n_predicates* and the corresponding defined indexes.

4 Experimental results

In this section we show some performance tests of the *fair predicate*, *equality-preferred* and *nonequality quarantining* algorithms with a variety of simulated loads. The performance tests discussed bellow were performed on a single-CPU Pentium workstation with a i686 CPU at 500MHz and 896MB RAM running Linux. We assume the existence of just one type of event with six optional attributes. The first three attributes have 200 possible values and can appear only in *equality* predicates. Their type is *string*. The other three attributes, of *numeric* type, have 5000 possible values and may appear in any type of predicate.

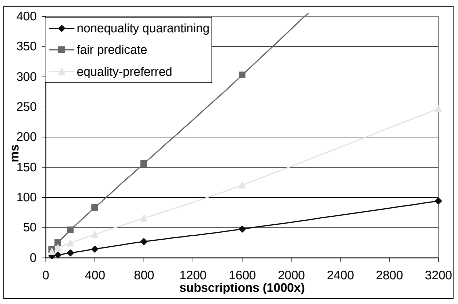
The subscriptions used in each test were generated as follows. Each attribute has a probability of 0.5 of being present in a subscription predicate. The first three attributes may only be present in an equality predicate while the last three may be presented in any type of predicate. There is also a probability p of generating an equality predicate for the last three attributes. The values associated to attributes in predicates were uniformly distributed over the corresponding attribute's domain.

In what concerns the generation of events, they are randomly created assuming the values each attribute can take are uniformly distributed over the attribute's domain. All attributes are considered optional and have a probability of 0.5 of appearing in the event.

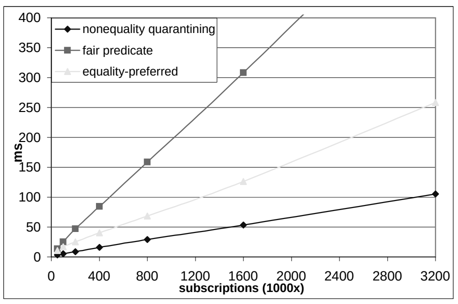
The generation of events and the execution of the matching algorithm are handled by two distinct processes running in the same machine. The first process sends a set of events to match to the second process. This one executes the matching algorithm and sends the result back to the first process. The result is represented by a list of matched subscriptions per event. The first process corresponds to the execution of a Java program, while the second corresponds to the execution of a K [12] program.

[2]which actually happens.

Figure 9.a shows the *performance (processing time)* of the several matching algorithms varying the number of subscriptions for a probability p equal to 0.5. The processing time also includes the communication time spent to send the events from the first process to the second one. Figure 9.b represents also the evolution of the *execution time* of each matching algorithm. The execution time is the sum of the processing and the communication time spent to send the matching result to the Java program. This communication time increases slightly with the number of subscriptions since the number of subscriptions matched by an event also increases with the number of subscriptions to verify. The *equality-preferred* and *nonequality quarantining* algorithms are the best ones, as expected. Among these two, the *nonequality quarantining* algorithm performs better because it reduces largely the number of sums made. Nevertheless, for a different configuration, the *equality-preferred* algorithm may be better. For example, if the *equality* predicates are not very selective, the number of partially verified subscriptions may be large and in this case the performance of the *nonequality quarantining* algorithm is worst than the performance of the *equality-preferred* algorithm. The performance of the *naive* algorithm is not shown in the these figures since it would be difficult to see the performance of the other algorithms. For example, for 400000 subscriptions, the naive algorithm spends almost ten thousand times more to process an event than the *nonequality quarantining* algorithm.



(a) Processing time



(b) Execution time

Figure 9: Processing time and execution time of algorithms: *fair predicate*, *equality-preferred* and *nonequality quarantining* for a probability p equal to 0.5.

Figure 10 shows the processing time of the *fair predicate*, *equality-preferred* and *nonequality quarantining* algorithms varying the probability p. The number of subscriptions used in this experiment was 800000. The *equality-preferred* and *nonequality quarantining* algorithms have the same performance when there is no *nonequality* predicate, since in this case both algorithms are equivalent. As can be seen in the figure, the performance of the algorithms depends on the ratio of *nonequality* predicates. The lower this ratio is, the better the performance is. This is due to the fact that the probability of an event satisfying a predicate is higher for a *nonequality* predicate than for an *equality* predicate. Therefore, the greater the ratio of the *nonequality* predicates, the more predicates are satisfied by an event and will have to be processed.

Figure 11 shows the performance of the *equality-preferred* (or *nonequality quarantining*) algorithm with the number of subscriptions to match, when each subscription contains just *equality* predicates. There is an initial *threshold* value that reflects the way the matching subscriptions are computed by the algorithm. In order to process an event, the algorithm has to initially determine the *equality* clusters whose *schema* is contained

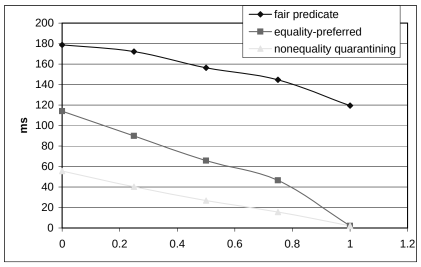


Figure 10: Influence of the number of *nonequality* predicates in the performance of the algorithms.

in the schema of the event. Since the subscriptions are generated randomly, all possible combinations of *equality* clusters are achieved for a very small number of subscriptions. Nevertheless, once this threshold value is achieved, the processing time of the algorithm increases slightly with the number of subscriptions, showing a logarithmic aspect. Also included in this threshold value is the communication time to send the events from the Java program to the K program.

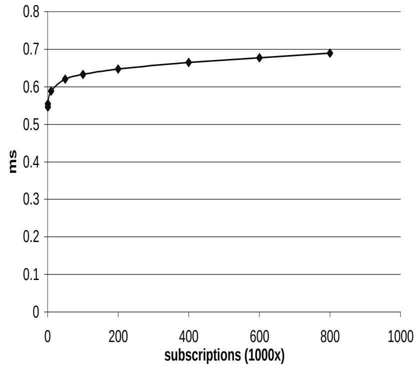


Figure 11: Performance of the algorithm *equality-preferred* (or *nonequality quarantining*) varying the number of subscriptions, where each subscription contains just *equality* predicates.

We also measured the time spent to add new subscriptions to the current set of subscriptions. With a current set of 3200000 subscriptions (and having p equal to 0.5), the average time spent to add one subscription is 32.2 milliseconds for the *nonequality quarantining* algorithm. *Equality-preferred* algorithm spends a similar time to update its internal subscription representation while *fair-predicate* algorithm is faster since it uses a simpler internal subscription representation than the other two algorithms.

5 Conclusions

In this paper we described the event model, subscription language and matching algorithm of our content-based event service: The event model permits an easy representation and integration of the information published on the *Web*; The subscription language is expressive; And the matching algorithm permits to handle a large number of volatile subscriptions with a high rate of events. To our knowledge, Le Subscribe is the first proposal for using an event notification service on a *Web* context. This context is characterized by an dynamic environment where the subscription language should be expressive, there is a large number of subscriptions and a high rate of events to process, subscribers can change their set of subscriptions at any time, and distinct publishers may produce similar kinds of information.

The matching algorithms developed by us apply a global optimization strategy to exploit predicate redundancy and predicate dependencies among subscriptions to reduce the number of predicate evaluations. Such a strategy is particularly efficient in the *Web* context where a lot of attributes have enumerated domains ranging over a limited number of values. The *fair-predicate* matching algorithm is *pure* predicate based. The other two result from optimizations applied to this one and are more efficient in time but less in space. The three algorithms can be easily adapted to be executed in a multi-processor environment for performance enhancement. In what concerns the *fair-predicate* algorithm, the clusters of predicates corresponding to each attribute-comparison operator pair can be distributed by the several available processors. Each processor will determine, for its assigned clusters, the predicates satisfied by the event to match and count afterwards the number of satisfied predicates per subscription. Finally, these several *counting* data structures will be collected by a last processor who will process them in order to compute the matched subscriptions by comparing the total number of satisfied predicates of each subscription with its number of predicates. A similar approach can be followed on the other two algorithms. The several *equality* clusters can also be deployed by the several available processors and be processed separately at each processor.

Similarly to the matching algorithm in NEONRules, our algorithms can also handle several types of events[3] simultaneously in a efficient way. If each event attribute appears in just one event type, it is not necessary to modify the algorithms since each *cluster* of predicates defined by the algorithms refers only to predicates of subscriptions related to the same event type. If this does not happen, each event attribute name can be concatenated with the corresponding event type name. Therefore, each event attribute appears in just one event type since different event types have different names. This extra processing (concatenation) can be made automatically during the pre-processing of the subscriptions (for the attribute names referred in the subscriptions) and the processing of the events to match (for the attributes of the events).

References

[1] Marcos K. Aguilera, Robert E. Strom, Daniel C. Sturman, Mark Astley, and Tushar D. Chandra. Matching events in a content-based subscription system. In *Eighteenth ACM Symposium on Principles of Distributed Computing (PODC '99)*, 1999.

[2] Amazon.com, Inc. *http://www.amazon.com*.

[3] G. Banavar, T. D. Chandra, B. Mukherjee, J. Nagarajarao, R. E. Strom, and D. C. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. In *International Conference on Distributed Computing Systems*, 1999.

[4] Arvola Chan. Transactional publish/subscribe: The procative multicast of database-changes. In *SIGMOD'98*, page 521, 1998.

[3]Considering the case where an event can only belong to a single event type.

[5] eBay Inc. *http://www.ebay.com*.

[6] Phil Bernstein et al. *The Asilomar Report on Database Research*, 1998.

[7] K J Gough and G Smith. Efficient recognition of events in distributed systems. In *Proceedings of ACSC-18*, 1995.

[8] R. E. Gruber, B. Krishnamurthy, and E. Panagos. The architecture of the ready event notification service. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems Middleware Workshop*, 1999.

[9] Eric N. Hanson, Moez Chaabouni, Chang-Ho Kim, and Yu-Wang Wang. A predicate matching algorithm for database rule systems. In *SIGMOD'90*, pages 271–280, 1990.

[10] SoftWired Inc. http://i gate.softwired.ch/products/ibus/. Developing publish/subscribe applications with ibus - technical white paper. 1999.

[11] IONA Technologies. *OrbixTalk http://www.iona.com/products/messaging/index.html*.

[12] KX SYSTEMS. *K USER MANUAL*, version 2.0 edition, 1998.

[13] New Era of Networks Inc. *NEONet http://www.neonsoft.com/products/NEONet.html*.

[14] New Era of Networks Inc. *NEONRules http://www.neonsoft.com/whitepapers/MQSIRules.html*.

[15] Yahoo! Inc. *http://auctions.yahoo.com*.