A CONTENT-BASED PUBLISH/SUBSCRIBE SYSTEM, ITS MATCHING ALGORITHM & APPLICATIONS

by

Vasilis P. Dialinos

A thesis submitted in partial fulfillment of the
requirements for the diploma of

Electronic and Computer Engineering

Technical University of Crete
Department of Electronic and Computer Engineering

Intelligent Systems Laboratory

Abstract

In the near future, the majority of human information will be in the Web. The searching, quering and retrieving information approach, while very efficient for static information, does not integrate well with the dynamic aspect of the Web. To address this issue, notification systems emerged, being able to capture the dynamic aspect of the Web by notifying users of interesting events. These may vary from stock markets updates, weather reports, availability of auction items etc. In this context, a notification system was proposed called Le Subscribe, upon which the Event Notification system we present in this work was based. The Event Notification system is able to cope with high rate of events, large number of user demands (post subscription, remove subscription) and efficiently notify users with very short delay. Moreover it provides a simple and comprehensive API enabling easy integration of existing Web applications with an event notification mechanism.

Contents

List of Figures

Chapter 1

Introduction

In this chapter a brief discussion of the systems currently available for quering and retrieving information will be made and how we contribute in this context.

1.1 Publish/Subscribe Systems

As pointed out in [8], the majority of human information will be on the Web in a few years. Besides the searching, quering and retrieving information approach widely being used, there is a need for systems being able to capture the dynamic aspect of the Web by notifying users of interesting events. These may vary from stock markets updates, weather reports, availability of airplane tickets etc. A tool that implements this functionality must be able to cope with high rate of events, large number of user demands (post new subscription, remove subscription) and must be efficient to notify the users with very short delay. In addition, it should provide a simple and comprehensive web application interface to enable users to easily add or remove subscriptions, view events matching their subscriptions, post events and modify the rules upon the events and subscriptions are posted. The system we present in this work is based upon *Le Subscribe*[22] and succesfully addresses these issues.

To better illustrate the issues that led *Publish/Subscribe* systems to emerge let us consider the set of auction sites in the internet(e.g. eBay[6], Amazon[14] or Yahoo![15]). Every day a large number of auction items are posted in each site. So users, in order to find an item of interest, must periodicaly access each site and repeat their queries, which may differ from site to site, to get the new interesting items. In a publish/subscribe system, a user posts one or more queries in the form of subscriptions. When an event matches a user subscription, the user is informed about the event. Moreover, in a pub/sub system each query is posted only once, the user does not have to deal with the heterogeneous quering mechanisms each site supports and when a query is not longer interesting, the user just removes the query.

1.2 Content vs. Subject-Based Systems

We can distinguish two kinds of pub/sub systems: *subject-based* and *content-based*. *Subject-based* systems, classify the events in groups or subjects and can only be filtered according to their group. A subject-based system would assign a group for each category. A publisher must assign each event with a group. Users can subscribe to groups, and be notified if an event belogs to groups of interest. For example, a subscriber can subscribe to be notified if a new auction item of type *car* is being put up.

Content-based systems are the emerging type of pub/sub systems, where events are evaluated according to *attribute-value* pairs, against subscriptions posted by subscribers. This way, subscribers can have more specific subscriptions. Example of such subscription is an auction item of type *car* and price lower than 1300€.

Comparing subject-based and content-based systems, the latter offer more subscription expressiveness, which of course has an impact in the complexity of the matching process. This complexity combined with a high event rate can severely degrade the matching efficiency. So systems devoted to handle high rate of events and a large number of subscribers have to face a trade of between the subscription language sophistication and matching efficiency.

1.3 Contributions

The system we present in this work implements efficiently the concepts illustrated in [8], which are

- A semi-structured event model which is well suited for the information published on the Web, and flexibe enough to support easy intergation of publishers.
- A subscription language which supports the most common queries.
- An efficient main memory algorithm to process on the fly a high rate of events with a large number of subscriptions.
- A simple and comprehensive API to enable easy integration of existing applications with an event notification mechanism.
- A demo web application to better illustrate the API usage.

1.4 Thesis Organization

This thesis is organized as follows. In Chapter 2 we briefly discuss releated work done in this context. Chapter 3 presents the event model and the subscription

language the system currently supports for event publication, subscription posting and schema extension. In Chapter 4 we discuss the problem of matching events with subscriptions and present an efficient solution to this problem. In Chapter 5 the programming languages, tools and technologies engaged in this project are described. Moreover, in Chapter 5 we present the application which was build for demostration purposes. In Chapter **??** the Event Notification API is documented and in addtition we describe how an application can be integrated with an notification mechanism. In Chapter 7 the implementation details of the matching algorithm are presented. In Chapter 8 we show the experimental results of perforance tests of the matching algorithm with a variety of work loads. Appendix A presents a sample XML schema upon which events and subscriptions are posted, Appendix B presents a sample XML log file holding subscriptions and matched events for subscribed users and finally Appendix C shows some sample XML requests with the corresponding DTDs.

Chapter 2

Related Work

The last few years the topic of event notification systems is gaining much popularity and attention. In this chapter we try to make a synopsis of the most important and influencing researches done so far. In the following sections SIENA[4], DIAS[16], P2P-DIET[**?**] and Le Subscribe[22] event notification systems will be discussed and see how these systems contributed in the context of publish/subscribe systems.

2.1 SIENA notification system

SIENA[4] is a distributed event notification service over a P2P server network. Clients connect to the network via access points provided by the servers, through which they can post events, subscriptions and advertisements. SIENA extends the standard publish/subscribe protocol with the use of advertisements, which reflect the client intension to publish particular kind of information. That information is valuable in routing subscriptions towards object of interest that intend to generate relevant notifications to that subscription.

The event notification service carry out a *selection* process, in order to determine which notifications are of interest to which clients. In addition, the *selection* process also provides means in controling the traffic in the P2P network. So filters can be applied in the notifications propagated resulting notification forwarding only to interested clients.

The architecture, the routing algorithms and the subscription language used by SIENA is the subject of discussion in the following subsections.

2.1.1 SIENA semantics

event notification In SIENA implementation is a set of triples of the form *(type, name, value)*. *Type* is the data type of the value, *name* is the attribute name and *value* is the attribute value. Arbitrary notifications are

composed from attributes selected by an restricted and well defined attribute set.

filters An *event filter* matches events by specifying a set of attributes and constraints over that attributes. In the set of available operators provided by SIENA, belong all standard comparison operators. An attribute $a = (type_a, name_a, value_a)$ matches an attribute constraint $\phi = (type_\phi, name_\phi, operator_\phi, value_\phi)$ if and only if $type_a = type_\phi \wedge name_a = name_\phi \wedge value_a = value_\phi$. When a matches ϕ, we signify it as $a \prec \phi$. When a filter is used in a subscription, all constraints must be matched with each subscription attribute.

Patterns *Patterns* can be matched against one ore more notifictions based on both their attribute values and on the combinations they form. Patterns syntactically are a sequence of filters.

adertisements An *advertisement* is relevant to a subscription if these to set of notifications have two non empty intersection.

unsubscribe/unadvertise An *unadvertise/unsubscribe* query cancels an advertisement, subscription respectively that it is covered by the unsuscribe/unadvertise filter.

2.1.2 Server topology

Before concluding to general peer-to-peer server topology, other solutions were considered.

Hierarchical client/server architecture The servers in this topology form a directed graph, where each server can have many incoming connections from servers, but only one outgoing connection to its own *master* server. A server with no *master* is referred to as *root*. This topology suffers from fault-tolerance issues, because if a master fails, all servers and clients connected to it, also fail. In addition servers high in the hierarchy take up much work load.

Acyclic peer-to-peer architecture In this topology, servers form a acyclic by-directional graph. Servers communicate with each other symmetrically as peers excanging subscriptions, events and advertisements. In such topology, algorithms should be aware and benefit from this property, but maintaining the property of acyclicity in a wide-area network is very defficult. Moreover, fault tolerance issues also arise (if a server fails, the network is split in two).

General peer-to-peer architecture Removing the constraint of acyclicity from a acyclic topology, we obtain a *general peer-to-peer architecture*, as shown

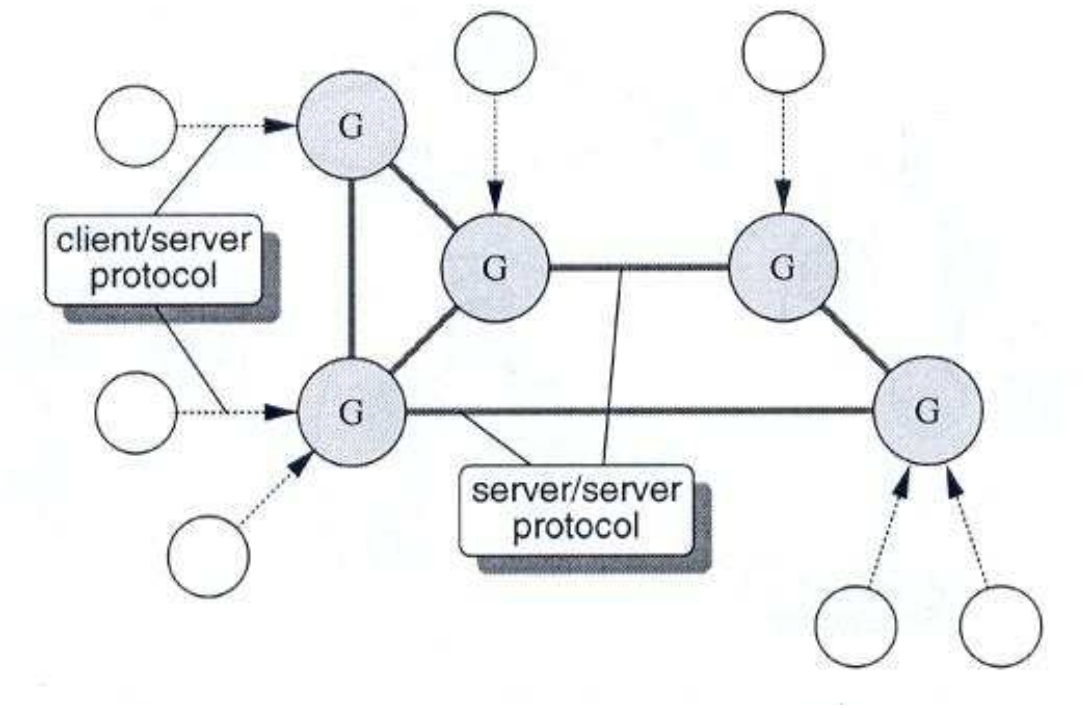

Figure 2.1: Siena general peer-to-peer topology, as illustrated in [4].

in Figure 2.1. The advantages of such topology are that less coordination needed. In addition the system is more fault tolerant and robust because more than one route exit between servers. In the other hand, special algorithms should be implemented and routes must be established using minimum spanning trees.

2.1.3 Routing strategies in Siena

The information routing inside the network is a very important issue, which can determine the robustness of the service. It is fair to say that all subscriptions must be matched with the notifications reaching the network. This could be done either by routing the notifications to subscriptions, or the opposite. Dude to the overwhelming number of notifications compared to subscriptions, propagating notifications to all network nodes is a solution least efficient. This issue was addressed in SIENA with the following techniques, as presented in [4].

downstream replication A notification is propagated as close as possible to the interested clients, and then is copied.

upstream evaluation Filters and patterns are applied the closest possible to the notification source.

subscription forwarding In an implementation that does not use advertisements, paths should be set by the propagating subscriptions, which a satisfied notification follows back to the interested client.

advertisement forwarding In implementations that do use advertisements, it is safe to send subscriptions only towards object of interest which have advertised notifications relevant to the subscription.

2.2 DIAS

DIAS[16] stands for *Distributed Information Alert System*, was developed in the context of the European project DIET[17, 10], and contributed in the area of event notification systems in the following ways. First, DIAS introduced a P2P agent architecture inspired by the event dissemination system SIENA[4] and second, formalized expressive data models and query languages suited for this context.

2.2.1 DIAS architecture

As illustrated in Figure 2.2 found in [16], the DIAS P2P network of middle-agents handle the notification of users about interesting events posted by the information providers, according to user subscriptions. Users and information providers interact with the P2P network through end-agents and resource-agents respectively.

User subscriptions are posted through the end-agent to the network of middle-agents, which handle the matching of the subscription and the forwarding to other agents. Subscription forwarding is done in an efficient way, e.g., if less general subscription found than those already posted then it is not forwarded. If more general subscrition is satisfied, less general are also satisfied. To decide whether a subscription is more or less *general* than another is called *entailment* or *subsumption* problem and it is handled efficiently with the data models DIAS supports.

2.2.2 DIAS data models and query language

Much of the work in DIAS was in building an expressive and formal subscription language. The data model used in DIAS is $\mathcal{AWP}$[16] which is based on text valued attributes and its query language is an extension of the query language of data model $\mathcal{WP}$.

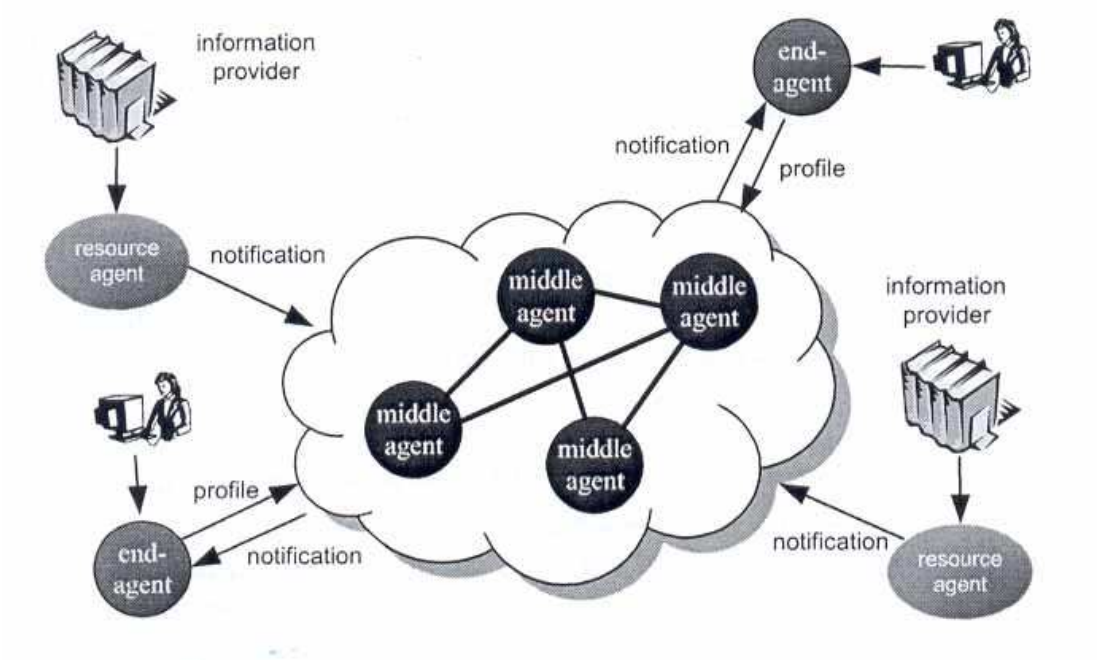

Figure 2.2: DIAS architecture, as illustrated in [16].

Lets now have a brief look on $\mathcal{WP}$ and $\mathcal{AWP}$ data models. The $\mathcal{WP}$ data model assumes that textual information is in form of free text and can be matched with *word patterns*. Word pattern may be any expression with conjunction, disjunction or negation of word patterns. For example,

$$Information \wedge alert \wedge \neg DIAS$$

is a word pattern. A very important feature of the $\mathcal{WP}$ data model are the *proximity operators*, which are represented with the $\prec_{[i,j]}$ symbol. The $\prec$ symbol specifies an ordering in word patterns, while i, j specify how close or far, in means of words, two word patterns separated by a proximity operator can be. So the following word pattern

$$Information \prec_{[0,0]} alert \prec_{[1,5]} DIAS$$

means that the word *Information* must be exactly before *alert*, while word *DIAS* must be in a distance of at least one word and most five after word *alert*.

Finally, $\mathcal{WP}$ defines the entailment operator $\models$. Let wp_1 and wp_2 be word patterns. Then wp_1 entails wp_2 ($wp_1 \models wp_2$) if and only if every text value that entails wp_1, also entails wp_2.

An example of this operator is that $Information \wedge alert$ entails $Information$. This feature is used in choosing the most general subscription in subscription propagation between middle-agents.

Now, we have seen the $\mathcal{WP}$ data model, we can define $\mathcal{AWP}$. Indeed, $\mathcal{AWP}$ is based on *attributes* with $\mathcal{WP}$ expressions as values. *Attributes* can be chosen from an *attribute universe* specific to each *notification schema*[1].

Over this notification schema, *notifications* can be posted in terms of *attribute, value* pairs. A *notification* is valid if it provides a binding for every attribute in the notification schema. A notification example over a notification schema with three attributes *author, title* and *abstract* would be

> {*(Author=Koubarakis), (Title=Boolean Queries with Proximity Operators for Information Dissemination), (Abstract=...)*}

Finally, a query over a notification schema could be an expression of the form $A \sqsupset wp$ or $A = s$, where A is an attribute from the attribute universe, wp any proximity free or not word pattern and s a simple text value. In addition, *conjunctions, disjunctions* and *negations* of queries are also valid.

For example the query

$$Author = \text{“}Koubarakis\text{“},$$

$$Title \sqsupset (Boolean \prec_{[1,5]} Proximity)$$

would be satisfied by the event presented above.

The P2P-DIET we present next, is a direct ancestor of DIAS and supports successfully all its features.

2.3 P2P-DIET

P2P-DIET[**?**] is a system based on the concepts of DIAS[16] and extends them in many interesting ways. P2P-DIET was build on top of the open source DIET Agents Platform, and successfully unifies *ad-hoc* and *continuous* query processing in P2P networks. With the term *ad-hoc* we are addressed to queries serviced by the network once when posted, while the term *continuous queries* specifies long lasting queries which notifies the interested party whenever resources of interest appear in the network. Now we are going to take a brief look in P2P-DIET architecture, functionality and query language.

2.3.1 P2P-DIET architecture

A high-level view of the P2P-DIET architecture is illustrated in Figure 2.3, as presented in [**?**]. The structural units of a P2P network are the nodes. In P2P-DIET are two kinds of nodes: *super-peers* and *clients*. All super-peers are equal

[1]Notification schema N is a *(A, V) pair, where* A *is a subset of the attribute universe and* V *a vocabulary.*

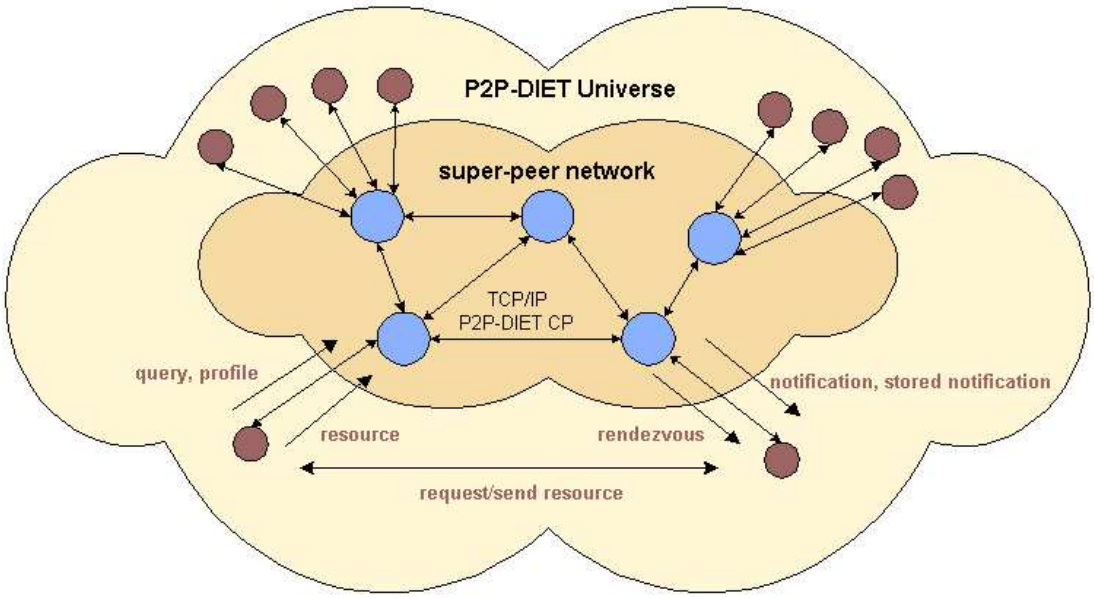

Figure 2.3: P2P-DIET architecture, as illustrated in [**?**].

in terms of responsibilities, so the node subset of the network composed only of super-peers, is a *pure* P2P network. Each super-peer share the work load of the network, serving a subset of clients and keeps indices of the client resources.

Resources are kept at client nodes and a client downloads directly from the resource owner node. Clients connect to the P2P network through a super-peer, where information about them are stored. Clients can migrate to an other access point or/and use dynamic IP.

Moreover, P2P-DIET has the ability to add or remove super-peers and in addition to recover when super-peers fail. Finally, security issues are also taken into account, with the use of message encryption and authentication between the peers.

2.3.2 Data models and query language

The P2P-DIET supports the $\mathcal{WP}$ and $\mathcal{AWP}$ data models and their corresponding query languages, as presented in [16]. The support of $\mathcal{AWPS}$ data model is currently under development.

$\mathcal{WP}$ is based on free text and its query language is based on the *boolean model with proximity operators*. Data model $\mathcal{AWP}$ is based on text valued attributes, and its query language is an extension of the query language of data model $\mathcal{WP}$. Finally the model $\mathcal{AWPS}$ extends $\mathcal{AWP}$ by introducing a similarity operator. Data models $\mathcal{AWP}$ and $\mathcal{AWPS}$, are attractive for the representation of textual

metadata due to their formality and expressive power.

2.3.3 Routing and query processing

The routing in the super-peer subnetwork is done using *minimum weight spanning trees*, which are rearranged whenever a new super-peer enters or leaves the network. Answers and notifications are unicasted to the interested super-peer through the shortest path connecting the super-peers. Now lets see the basic functionalities provided by the P2P-DIET:

Ad-hoc quering P2P-DIET supports the typical *ad-hoc query* senario, where a client C posts a query to its access point AP. AP multicasts this query using the minimum spanning tree to all the super-peers of the network. Answers are posted back to the AP through the shortest path connecting each super-peer with with AP.

Continuous quering The selective information dissemination senario is discussed here. A client C posts a *continuous query* to its access point AP. AP forwards the query to other super-peers. When a resource satisfies the forwarded query, the answer travels back to AP through the path set by the query propagation.

Lets now see some additional functionalities P2P-DIET supports:

joining the network A new client, in order to enter the network must first aquire the IP-address and public key of a convenient client. Then the client sends an encrypted `new-client` message containing its public key, identifier and IP-address. After the message decryption and processing the client can be securely identified.

Publish resources After clients joining the network, may publish resources by sending *resource metadata* to the access point using the data models supported. Resources metadata are not multicasted, as they are matched with incoming queries.

Disconnecting A client upon disconnect, notifies the super-peer used as access point. The super-peer keeps the resources metadata of the disconnected client for a specific period of time. On reconnect, the client tries to connect to the access point connected the last time, unless the user requests a *migration* to a different access point.

Processing notification and answers A client, in order to download from the resource client, must know his IP-address and the local path of the resource. This information is stored in the notification of the resource the interested client receives. More useful information are also stored in the notification,

such as the resource client identifier and its status when the notification was produced. If the resource client is on-line and the client which received the notification chooses to download the resource immediately, then is directly connected to the resource and downloads it. In other case (resource not currently available or interested client downloads later), the interested client must query his super-peer the current IP-address of the client with the resource, using the resource client identifier.

Stored notifications and rendezvous For the reason that clients are not on-line all the the time, there must be a mechanism to ensure that the clients always receive their notifications. This is done with the use of a *notification storage* in the access point of the client notified. When the client logs in, is informed about its notifications. Moreover, the *rendezvous* functionality enables clients to exchange resources without actually being online at the same time. This is accomplished with the use of temporary storage in super-peers called *rendezvous directories*, where resources are temporary stored.

Fault-tolerance The system is self-organized when a fault on a super-peer or on a client occur. A super-peer failure is traced when the super-peer with the problem can not respond to the `are-you-alive` messages send periodically by other super-peers. In this situation, the minimum-spanning trees connecting each super-peer with another, are rearranged. Identical messages are also sent periodically to the clients.

The P2P-DIET is an query and event notification system which is scalable, fault tolerant and location address indepentent, so it can work efficiently in the real world, the internet.

2.4 Le Subscribe

Le Subscribe[22] is the system upon our system is built. The 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 the event notifications.
- An efficient matching algorithm for processing in real time which can handle a large number of volatile subscriptions (several millions) and supports high event rates (several millions per day).

- Simple interface for publishing and subscribing which enable an easy integration of the system in the Web. The system supports both HTTP protocols and Java RMI.

The three matching algorithms developed for Le Subscribe, apply a global optimization strategy to exploit predicate redudancy 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[2] is *pure* predicate based. The other two algorithms, *equality-preferred* and *nonequality quarantining* result from optimizations applied to the first one and are more efficient in time but less in space.

2.5 Other publish/subscribe systems

Publish/subscribe systems are divided into major categories, *subject-based* and *content-based*. Examples of subject-based systems are OrbixTalk[7] and TIB/Rendezvous[5]. In subject-based systems, events are classified by groups and can be filtered only according to their group.

Content-based systems are the emerging type of publish/subscribe systems where events are filtered according to their attribute values. Content-based systems, other than Le Subscribe, are Gryphon[1], NEONet[20] and READY[11] and publish/subscribe mechanisms integrated in commercial DBMS products like Oracle8i, SQL Server 7.0, or Sybase. The cost of this gain in excpressiveness is an increase in the complexity of the matching process: the more sophisticated the constructs, the more complex and time consuming the matching process.

The subscription languages of Gryphon, and NEONet are quite similar to Le Subscribe language. Their matching algorithm do not exploit predicate redundancy nor dependencies. The READY system[11] has a more expressive subscription language supporting grouping constructs, compound event matching and event aggregation. Its matching algorithm uses only local optimizations unlike Le Subscribe which intensively exploits global optimization opportunities and predicate redundancies. Commercial DBMS products use SQL as their subscription language, and these products are designed for contexts where the number of subscriptions is relatively small, as might occur in the context of enterprise application integration.

Hanson *et al* proposes a matching algorithm in [12]. During a pre-processing phase, the algorithm chooses the most selective predicate for each subscription and places it in an index (IBS tree) associated with the predicate's attribute. During the processing of an event, the algorithm first computes the set of subscriptions whose most selective predicate is verified and then checks the remaining

[2]similar to *Counting algorithm* we implemented.

predicates of each selected subscription in a naive way.

Chapter 3

Event Model & Subscription Language

In order to integrate the system to handle publications from different users, it must solve the problem of heterogeneity of information sent by different publishers. That is accomplished by having publishers publish their events with respect to an event schema, and subscribers post queries using an well-defined subscription language. So the system must define:

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

3.1 Integrated event schema

The purpose of the integated event schema, is to provide the model over which events are published. As illustrated in Le Subscribe[22] an Integrated Event schema[1] S is a sixtuple($\mathcal{A}$, $\mathcal{ET}$, $\mathcal{D}$, *DOM, EVENT_TYPE*, $\mathcal{E}$). The semantics associated with each symbol of the *sixtuple* are the following:

$\mathcal{A}$ This symbol represent the attribute universe of the IE schema. Each attribute is identified uniquely by its name. In the set of attributes $\mathcal{A}$ defines exists an distinct attribute *event_type* with a special meaning.

$\mathcal{D}$ This symbol represents the set of attribute domains. Each domain can be either of *string, enumerated* or *numeric* type and is associated with a set of operators from those available for each data type. For example *numeric* domains support all standard comparoson operators[2], *enumerated* domain

[1] From now on we are going to address to Integreted Event schema, as IE schema.

[2] Meaning $=, \neq, \geq, >, \leq, <$.

supports *equality* and *non equality* comparison operators over a pre-defined value set. Finally, a *string* operator supports *equality*, *non equality* as well as *contains* comparison operator.

$\mathcal{DOM}$ This symbol represents the function which associates each attribute with its domain. The $\mathcal{DOM}$ set holds an entry for each attribute with two identifiers, the attribute name and the attribute domain.

$\mathcal{ET}$ Represents the domain of the distinguished attribute *event_type*. Its domain exists in the domain set of $\mathcal{D}$ and its domain type is of *enumerated* type.

$\mathcal{EVENT\_TYPE}$ This symbol represents the set of values over which *event_type* is ranging.

$\mathcal{E}$ Represents the function which associates each value of $\mathcal{EVENT\_TYPE}$ with the corresponding *event schema*. An *event schema* is a set of triples of the form (A, n, u), with $A \in \mathcal{A}$ and n, u two annotations with the first ranging over values in {mandatory, optional} set while the latter over values in {unique, multiple}.

This concepts will be better understood with the following example: Suppose an item of type *antique* which is described by three mandatory attributes *price, period* and *quantity* and an item of type *car* which is also described by three mandatory attributes *brand, cc* and *price*, while has one optional attribute *color*. We observe that attribute *price* is in common for the two items. The IE schema in this case is the following:

- $\mathcal{A}$ = {price, period, quantity, brand, cc, color, event_type}
- $\mathcal{ET}$ = {antique, car}
- $\mathcal{E}$(car) = {(price, mandatory, unique), (brand, mandatory, unique), (cc, mandatory, unique), (color, optional, unique)}.

To make the concept of publication of an event/subscription over an event schema solid and formal lets see the following definitions, as given in [22].

Attribute set, Mandatory set, Unique set 3.1.1 *Let* $S = (\mathcal{A}, \mathcal{ET}, \mathcal{D}, \mathcal{DOM}, \mathcal{EVENT\_TYPE}, \mathcal{E})$ *be an IE schema and* e *an element of* $\mathcal{ET}$*. Then the* Attribute set *for* e *is defined as the set of attributes present in* e event schema. Mandatory set *and* Unique set *are subsets of the* Attribute set*, only containing attributes of* $\mathcal{A}$ *which have a* mandatory *and* unique *notations respectively. For Example, the Mandatory set for* car *is constituted by attributes price, brand and cc.*

View over an IE schema 3.1.1 *A view $\mathcal{V}$ over IE schema S = ($\mathcal{A}$, $\mathcal{ET}$, $\mathcal{D}$, $\mathcal{DOM}$, $\mathcal{EVENT\_TYPE}$, $\mathcal{E}$), is a pair $\{\mathcal{E}, \mathcal{R}\}$, with $\mathcal{E}$ a set of event types occuring in $\mathcal{ET}$ and $\mathcal{R}$ the conjuction of* (A, n, u) *triples found in the event schema associated with each event type of $\mathcal{E}$.* (A, n, u) *triples must satisfy the following rules:*

1. For each mandatory attribute A of each $e \in \mathcal{E}$ must be a *(A, n, u)* triple in $\mathcal{R}$,
2. *(A, n, u)* triple occurs in $\mathcal{R}$ only if A occurs in the attribute set of at least one e of E,
3. every *(A, n, u)* triple must be unique in $\mathcal{R}$ over its attribute. Mandatory and multiple annotations have priority over the optional and unique ones.

Event instance 3.1.1 *Let S = ($\mathcal{A}, \mathcal{ET}, \mathcal{D}$,* 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 rules:*

Rule 1 *ei* contains the pair *(event_type, E)*

Rule 2 for each element *(A, n, u)* of $\mathcal{R}$, *ei* contains a pair *A, V* with V subset of *A* value domain. If *u* has the *unique* value, then *V* is a singleton.

Rule 3 Only pairs satisfying rule *R1* or *R2* occur in *ei*.

Suppose the following example: A publisher posts an event of an antique car. In this case, the event belongs to both event types. Consider the following events e_1={*(event_type,(antique, car)), (price, 1300), (period, 1970), (quantity, 1), (brand, cooper), (cc, 1000)*} and e_2={*(event_type,(antique, car)), (price, 3500),(period, 1970), (brand, audi), (cc, 1200)*}. Event e_1 satisfies all the rules while e_2 does not satisfy 2^{nd} Rule (and obviously the 3^{rd}), since the mandatory attribute *quantity* is missing in e_2.

3.2 Integration model

The IE schema, as illustrated in [8], provides an unified representation of the publications issued by distinct clients. With the term *integration model* we mean the publication language over the IE schema, as well as the IE schema extension capabilities our system provides to clients.

Publication language 3.2.1 *In the system we present in this work, a publication is valid only if it is expressed in the form of an* event instance *over the IE schema.*

IE schema modification rules 3.2.1 *As proposed in [8], the system we present allows publishers to extend the IE schema using four simple* modification rules. *Lets consider the* $S = (\mathcal{A}, \mathcal{ET}, \mathcal{D},$ dom, $\mathcal{E})$ *IE schema*

Domain creation rule 3.2.1 *makes a new domain entry in* $\mathcal{D}$ *and specifies the comparison operators over this domain, with respect to the set of available operators the domain type supports.*

Attribute creation rule 3.2.1 *adds a new attribute in* $\mathcal{A}$ *and makes a new entry in the* $\mathcal{DOM}$ *set associating the new attribute with a value domain.*

Event schema extension rule 3.2.1 *extends the event schema of an event type* e *by adding a triple (A, optional, u) to the event schema of* e. A *is an attribute not previously in the event schema.*

Event schema creation rule 3.2.1 *creates a new event type by adding a new entry in the* $\mathcal{ET}$ *set and extends the* $\mathcal{E}$ *function with the new event type schema.*

> **Note** The modification rule effect is global, meaning that it does not only effect the publisher who modified the schema, but all the publishers in general. For this reason, an event schema can only be extended with optional attributes, which do not further restrict the schema.

3.3 Subscription language

A subscription S is defined as a conjunction of elementary predicates. A predicate is a triple of the form (A, OP, V), with A the name of the attribute, OP one operator from the set of operators associated with the attribute domain and V a value from the attribute value domain.

Event instance matches subscription 3.3.1 *Given a subscription* S *and ie an event instance ,ie matches* S *if the following contitions are met:*

Rule 1 *If* e *provides a binding for every attribute in* S.

Rule 2 *All predicates are true with respect to this binding.*

For example, given the subscription [(*event_type contains car*) *and* (*price* $<$ 1300) *and* (*brand equals mini*)] and the *event instances* ei_1={*(event_type, (antique, car)), (price, 900)*}, ei_2={*(event_type, (antique, car)), (price, 900), (brand, audi)*}, ei_3={*(event_type, (antique, car)), (price, 1100), (brand, mini)*}. Applying the rules of 3.3.1, we can see that only ei_3 matches the subscription, while ei_1 does not satisfy the 1^{st} rule and ei_2 does not satisfy the 2^{nd}.

we can see that only ei_3 matches the subscription.

Currently, the system we developed provides all standard comparison operators over *numeric* domains, for *enumerated* domains provides *equality* and *non equality* comparison operators over pre-defined value set and for *string* domains *equality*, *non equality* and the distinguished *contains* operators are available. A *contains* predicate *(A, contains, V)* is true if its value V occurs in the set of values associated with the attribute A. For example $[(event\_type\ contains\ car)\ and\ (price < 1300)]$, describes an event of any car costing less than 1300€.

Disjunction of the elementary predicates and hierarchical domain are not yet supported.

Chapter 4

Matching Algorithm

The matching problem can be formulated as the following question:

> Given an event e and a set S of subscriptions, how can we found efficiently the subscriptions of S satisfied by e?

A graphical representation of this concept is illustrated in Figure 4.1. In this chapter, an efficient algorithm is presented to address this issue. This algorithm is resposible for matching every single event with all the subscriptions, one at a time. This algorithm was presented in [8], and is called *Counting Algorithm*.

4.1 Reformulation of the matching problem

A naive approach to this problem, would be the testing of each subscription against each event. This *naive* algorithm, as presented in [22] is shown in 4.2.

Taking a closer look to the algorithm, we see that in line 6, the fuction `pred_match` is responsible for matching all predicates of each subscription with the incoming event. During the processing of one event, the calls made to `pred_match` function are equal to the number of subscriptions multiplied with the number of predicates found in each one. In a real world senario, we expect *naive* algorithm being a hardly satisfactory solution, since the *naive* response time is proportional to the number of subscriptions.

The main drawback of the *naive* approach is that it does not consider the predicate redudancy among subscriptions to reduce the number of predicate evaluations. Such strategy is quite efficient in the Web context where a lot of attributes have enumerated domains ranging over a limited value set. So focusing on predicates instead of subscriptions can result in an optimazation to the efficiency of the algorithm, proportional to the redudancy of predicates.

Additional optimazation can be applied to the algorithm efficiency, considering the predicate dependencies, which means that if predicate $(price,\ \leq,\ 10)$ is verified by an event then the predicate $(price, \leq, 20)$ is also verified; but if

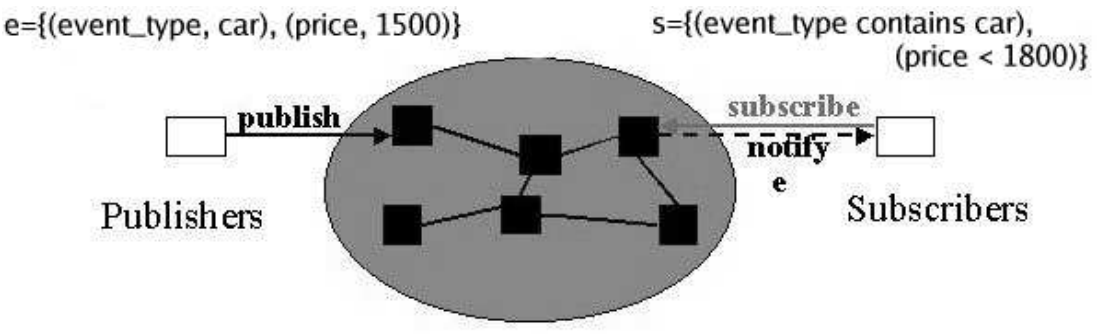

Figure 4.1: Graphic representation of matching problem

predicate ($price$, $=$, 10) is verified, predicate ($price$, $=$, 20) cannot be verified. So keeping predicates suitably ordered can result to further deduction of the proccesing time of the mathcing algorithm.

So the problem now is to compute the set of satisfied subscriptions from the satisfied predicates. A solution proposed in [22], is the *Counting algorithm* which was also used and evaluated by us in this work.

For every algorithm, there is always a pre-processing phase, responsible for representing the data sent to the algorithm in a suitable form. In the following sections the pre-processing and the matching algorithm is discussed.

4.2 Pre-processing phase

In this phase, the subscriptions are decomposed in their predicates and two hash tables are maintained. The first hash table clusters the predicates by comparison operator and value, while the second hash table is an association table which makes the correspondence between each predicate and the subscriptions it appears in.

each predicate is stored just once

Each hash table cluster is kept sorted in order to achive a further decrease in predicates evaluation (if predicate ($price$, $\leq$, 10) is verified by an event then the predicate ($price$, $\leq$, 20) is also verified).

4.3 Counting algorithm

The *counting algorithm* (Figure 4.3), as illustrated in [22] is a three step algorithm. The first step of the algorithm (lines 4-9) computes the satisfied predicates applying each event predicate to all the subscription predicates, which are stored

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

Figure 4.2: Naive algorithm

in the predicate hash table maintained in the pre-processing phase. *eqpredmatch, lesspredmatch, lesseqredmatch, greaterpredmatch, greatereqpredmatch* and *containspredmatch* functions compute the *equality*,*less than*, *less than equal*, *greater than*, *greater than equal* and *contains* predicates satisfied by a given event, respectively.

In the second step of the algorithm (lines 11-15), for each satisfied predicate p found in the previous step and for each subscription s containing p, a hit count is applied in s.

finally, in the last step of the algorithm (lines 17-21) the number of each subscription predicates is compared with the hit counter specific to each subscription. If these two numbers are equal, means that all subscription predicates where satisfied by the event.

Taking a closer look at the functions of the first step, we have the *eqpredmatch* function which searches, for each event's attribute, the predicate with the same

value as the attribute value in the *equality* cluster associated to the attribute.

The *lesseqpredmatch* searches, for each event's attribute, the predicate with the greatest value less than or equal to the attribute value in the *less than equal* cluster associated to the attribute. After having found this predicate, we know that all predicates that are placed before (while predicates are stored in increasing order) are also satisfied by the event. The *greatereqpredmatch* function is similar to *lesseqpredmatch*.

The *lesspredmatch* function searches, for each event's attribute, the predicate with the greatest value less but not equal to the attribute value. Because the predicates are kept ordered with a binary algorithm, all the predicates below the one found are also satisfied. The *greaterpredmatch* function works in a similar manner.

The hash tables maintained to store the subscriptions predicates and the binary search algorithms used for predicate quering in the hash table clusters, guarantee the efficiency of the algorithm, as we will see in Chapter 8.

counting_algorithm(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 $satisfied\_preds \leftarrow satisfied\_preds \cup lesseqpredmatch(e)$

7 $satisfied\_preds \leftarrow satisfied\_preds \cup greatereqpredmatch(e)$

8 $satisfied\_preds \leftarrow satisfied\_preds \cup greaterpredmatch(e)$

9 $satisfied\_preds \leftarrow satisfied\_preds \cup containspredmatch(e)$

10 // Step 2 - Applies hits on subscription for each satisfied predicate

11 ***foreach*** $p \in satisfied\_preds$ ***do***

12 ***foreach*** $s \in pred\_to\_subs[p]$ ***do***

13 $hitcounts[s] \leftarrow hitcounts[s] + 1$

14 ***endloop***

15 ***endloop***

16 // Step 3 - Counts the satified predicates per subscription

17 ***foreach*** $s \in S$ ***do***

18 ***if*** $hitcount[s] = \#subscriptions[i]preds$ ***then***

19 $matched \leftarrow matched \cup s$

20 ***endif***

21 ***endloop***

Figure 4.3: Counting algorithm

Chapter 5

System Implementation

In this chapter, the system implementation details are presented. Namely, the programming laguages and tools used and the technologies engaged in this project. Moreover, the implementation choices made and why preferred over other solutions are discussed. The algorithm implementation is presented in chapter 7.

5.1 System architecture

As we can see in the Figure 5.1, the system we developed is a centralized client/server application. Clients establish TCP/IP connections with the server and send XML requests, while the server replies with XML responses to clients.

5.2 Server-side implementation

The server-side of the application, is the part responsible for handling the client requests. This task is managed by a daemon application, the *Notification Daemon*. Notificication Daemon correspons to the excecution of a Java program.

5.2.1 Thread pooling

Notification Daemon creates a TCP/IP server socket and waits on a **listen()** command to accept user requests. When a request comes, a new TCP/IP connection is established between the server and the request publisher, functionalities provided by the Java built-in **java.net** package for network support. In every user request, a new thread from the *Thread Pool* wakes and continues the request handling. The thread pooling techinique is very popular among web applications[1] because the memory and time consuming thread intitialization is done once in server startup, during while a number of threads are initialized and suspended.

[1] or client/server applications, more generaly speaking

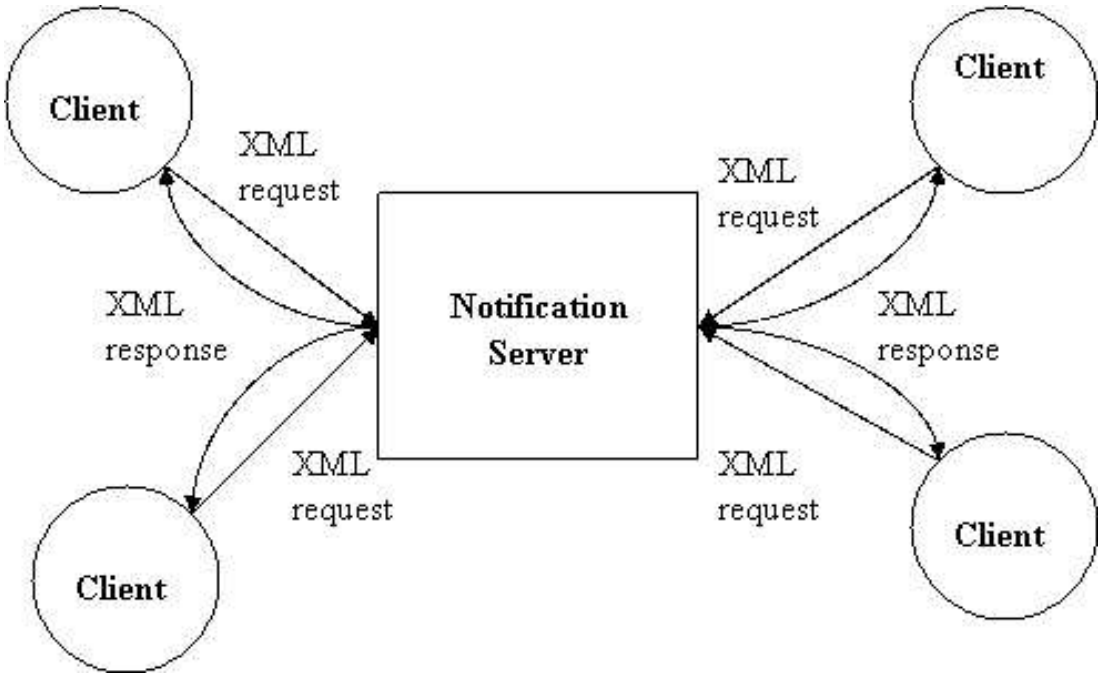

Figure 5.1: System architecture representation

When a new request comes to the server, a sleeping thread wakes and handles the event, without the overhead of a new thread initialization. Figure **??** is a simple representation of this concept.

5.2.2 XML-based communication

The communication between the server and the client is done with the use of XML documents. XML is a markup language heavily used in the Web. XML among others uses, can represent data and force some restrictions on the document format, with the use of DTDs. DTD is a set of restrictions which can be embedded in the XML document or, as in our case, can be linked externally.

In our case, XML is used by the client-side application (publishers, subcribers) in order to have a uniform representation of the requests posted to the server. In this manner, heterogeneity of similar information sent by different users can be masked. This enables subscribers to specify their requirements without having to face with heterogeneous information. Moreover, XML-based communication specifies a well-defined common interface, which enables easy integration of the system in a distributed environment.

The DTDs in the system play a two-fold role. First, validate the XML requests to ensure that are well formatted. That is that XML document follows some rules

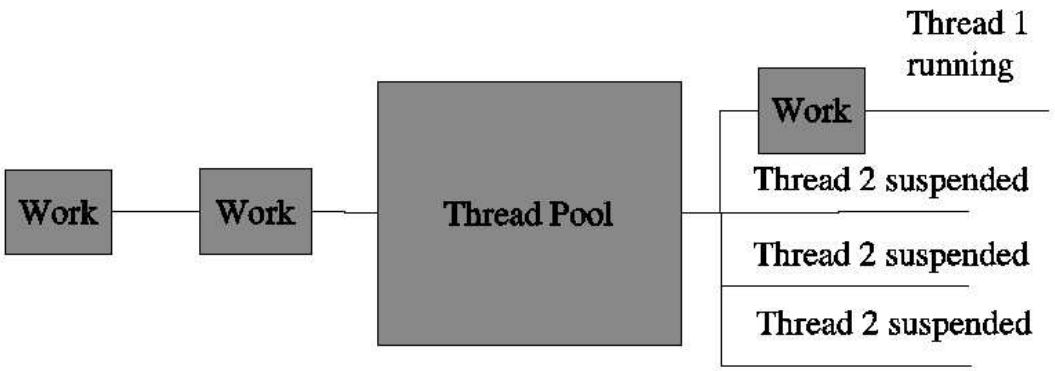

Figure 5.2: Thread pooling technique

in order to be complete and valid. And second, DTDs perform a parsing in the data send. Taking into consideration that the IE schema upon requests are send, changes dynamically with the *schema extension rules*[2], DTDs must be themselves dynamically redeclared whenever a schema modification occurs.

DTDs reside on the server-side and in order a request to be valid, must be externally linked with the appropriate DTD, depending on the request type. The concept of DTD validation and parsing becomes more comprehensible with the example of figures 5.2.2, 5.2.2. This example considers the IE schema of appendix A. The figures show how an event may look like in XML representation and how the DTD, over the event publication schema, would be formatted. The first of the following figures, represents the event e={*(event_type,(antique, car) , (price, 1300), (period, 1970), (quantity, 1), (brand, cooper), (cc, 1000)*}. Taking a closer look at the first figure of the example, between the `<EVENT_TYPE>` and `</EVENT_TYPE>` intentifier are the two event types of e and the attributes are enclosed between `<ATTRIBUTES>` and `</ATTRIBUTES>` intentifiers[3] . The distinct pair *(contact, sbile@intelligence.tuc.gr)*, which is a parameter in the `<PUBLISH_EVENT>` root element holds the information required in order to get in touch the interested party

[2]Covered extensively in section 3.2.

[3]from now on, we going to address to these identifiers as element, which is the proper XML terminology.

(i.e. subscriber) with the event publisher and does not have to be representet somehow in the IE schema.

The second figure illustrates the the *Document Type Definitions* document (DTD), based on the sample schema of appendix A, over the event publication schema. Since it is beyond the scope of this thesis to get in much details about DTD usage, we are going to have a brief discussion about some lines in the DTD document of the figure 5.2.2. So, the first line of the DTD, `<!ELEMENT PUBLISH_EVENT ( EVENT_TYPE , ATTRIBUTES )>` specifies that inside the `<PUBLISH_EVENT>` scope must be exactly two elements, `<EVENT_TYPE>` and `<ATTRIBUTES>`. In the third line we see `<!ELEMENT EVENT_TYPE ( VALUE+ )>` which means that inside scope designated as `<EVENT_TYPE>` must be at least one element named `VALUE` and nothing more. The + symbol has the regular expression semantics. Finally, in fourth line of the document is `<!ATTLIST VALUE type ( antique | furniture | surf-board | car | painting ) #REQUIRED>` which means that the elements `<VALUE>` attribute type, can take a value from the enumerated set *antique, furniture, surf-board, car, painting*. In the first two examples we saw DTDs apply on the XML format, while in the last example DTD was applied in the data of the document also.

> **Note** that the DTD check imposed on the XML documents may be redundant, if we consider that the requests have been previously checked by the HTML form[4] used by the publisher with the help of JavaScripts. But these checks is important to be performed since the system must be extentible to work with other clients and possibly other servers.

5.2.3 Server implementation summary

To sum up, events are handled by the Notification server daemon which has the following characteristics

- Multithreaded Java Application.
- Implements thread pooling technique to minimize response time.
- XML-based communication over the IE schema.
- DTDs sees that the XML documents send are well-formatted.

[4] we will see later the details.

```
An XML event
<PUBLISH_EVENT contact="sbile@intelligence.tuc.gr">
   <EVENT_TYPE>
      <VALUE type="antique" />
      <VALUE type="car" />
   </EVENT_TYPE>
   <ATTRIBUTES>
      <ATTR type="price">1300</ATTR>
      <ATTR type="period">1970</ATTR>
      <ATTR type="quantity">1</ATTR>
      <ATTR type="brand">cooper</ATTR>
      <ATTR type="cc">1000</ATTR>
   </ATTRIBUTES>
</PUBLISH_EVENT>
```

Figure 5.3: Sample XML event publication

5.3 Request handling thread

So far, we saw how the Notification server accepts requests and incorporates new *request handling threads* to handle each one. It is also discussed that the requests arriving to the server is the XML representation of those posted by the user. In this section we will see in details the course of actions followed by each thread in order to handle the request.

The first thing that has to be done, is to represent the XML document in a way convenient for reading and manipulation. This functionality is provided by the JDOM API[18]. JDOM uses the **jdom.org.input. SAXBuilder** class which implements a validating SAX[19] parser. The SAX parser reports parsing events (such as the start and end of elements) directly to the application through callbacks and does not have to build an in-memory tree structure of the XML document[5], which put a great strain on system resources, especially if the document is large.

So after the XML request has been read and validated by the parser, we are ready to access it's elemens values and attributes in which the request information are stored. A request may be an *extensions schema request*, an *event request* or a *subscription request*. The information concering the category each request belongs is specified by the name of the document's root element, which is at the first line of the document[6]. If the document is in any way malformed an **org.jdom.JDOMException** is thrown and the document receives no further processing.

[5] as DOM does

[6] after the XML tag of course

```
DTD over event schema
<?xml version="1.0" encoding="UTF-8"?>}
<!ELEMENT PUBLISH_EVENT ( EVENT_TYPE , ATTRIBUTES )>
<!ATTLIST PUBLISH_EVENT contact CDATA #REQUIRED>
   <!ELEMENT EVENT_TYPE ( VALUE+ )>
   <!ELEMENT VALUE EMPTY>
      <!ATTLIST VALUE type ( antique | furniture | surf-board |
         car | painting ) #REQUIRED>
   <!ELEMENT ATTRIBUTES ( ATTR* )>
      <!ELEMENT ATTR (#PCDATA)>
         <!ATTLIST ATTR type ( price | period | quantity |
            furniture_category | material | year | volume | cc |
               brand | artist | size | event_type ) #REQUIRED>
```

Figure 5.4: Sample event publication DTD

5.3.1 Extention schema request handling

In this category falls every request concerning the extension of the schema. An extension schema request may be either of the following: *add domain request* , *add attribute request* , *extent event schema request* or *add event request*. Sample schema extension requests are presented in appendix C. When an extension schema request is about to be processed no other thread must be reading or writing the IE schema. In order to achieve this we have to synchronize the processes to have serial access to the IE schema. IE schema is a XML document parsed and manipulated with the use of SAX (sample IE schema lays in appendix A).

add domain request During the processing of an *add domain request* the course of action followed are discussed here. At first, the handling thread acquires a write lock on the *IEschemaLock* object, which manages the access to the IE schema. When the lock is acquired, the thread is ready to add an new element `<DOMAIN>` entry in the parent element `<D>`. This entry holds the information concerning the domain type and operators of the new domain. In addition, there is a unique id number assigned to each new domain. When the IE schema update is over, the *IEschemaLock* is released and a new lock, *DTDlock*, is aquired to serialize access to the DTDs documents. This lock prevents other running threads from either reading or writing any DTD. Then the thread updates the *add domain* and *add event* DTD, which are the DTDs affected by the new domain insertion. *DTDlock* is released when DTD update is finished and thread suspends.

add attribute request With the same manner as above *add atribute request* is

handled. First of all, the thread aquires a write lock on the *IEschemaLock*, if the object is locked by other thread, suspends. When lock aquired, two new entries are inserted in the schema. One element `<VALUE>` entry in the parent element `<A>` and one `<ATTR_DOM>` entry in the parent element `<DOM>`. The first is an entry in the attribute repository, while the second is the association of the attribute with it's value domain. If the attribute is already in the attribute repository, a **thesis.exception.SameAttributeInSchemaException** is thrown.
Then *IEschemaLock* is released, thread enters the *DTDLock* monitor and DTDs for *extent event schema*, *add event*, *publish event* and *publish subscription* are updated. When done, thread leaves monitor, suspends and waits in thread pool to handle another request.

extent event schema request Likewise, after entering *IEschemaLock* monitor, the thread updates the IE schema by associating the attributes with the corresponding event type schema. During this operation three types of exceptions may be thrown. A **thesis.exception.NoSuchAttributeException**, if one of the attributes specified in the request is not in the IEschema repository, a **thesis.exception.SameAttributeInSchemaException**, if an attribute is already in the event type's schema and a **thesis.exception.Event TypeNotFoundException**, if no such
event type as one about to be extented exists in IE schema. If no exception is thrown after this operation, the IE schema must contain in the element `<E>`, where all event types are hold, in the `<E_TYPE>` element corresponding to the event type the extension is done, the new `<E_ATTR>` element (or elements). which represent the new element (or elements) associated to this event type schema. After that, *extent event schema* DTD is created and thread suspends[7].

add event request Adds a new request in the schema, by adding an element `<VALUE>` entry in the parent element `<ET>`, where all event type names are stored and updates the content of element `<E>`, which maps the event types with the attributes appearing in their schema. In this context an **thesis.exception. SameEventInSchemaException** or a **thesis.exception. NoSuchAttributeException** may be thrown, if the event type schema about to be added is already in the IE schema or one of the new event type's attributes are not in the attribute repository, respectively. After that, the DTDs for *extent event schema*, *add event*, *publish event* and *publish subscription* are created and then thread suspends.

[7] to avoid repetition, monitor entering and leaving will not be discussed further

5.3.2 Subscription request handling

As we will see in a consequent chapter, all subscription's predicates are kept in an in-memory structure called *PredicateClusters*. In addition, another structure is maintained in main memory called *PredToSub*, that holds the mapping between the predicates and the subscriptions they appear in. The access to these structures must be managed in a read/write manner. Meaning that concurrent access to these structures for reading must be allowed, but only one thread must update their content at a time. This structures are kept in the native side of the system, meaning that they are maintained and processed by the C++ code. To this part of subscription handling we are reffering to as the *preprocessing* of the matching algorithm.

Before XML subscriptions are fed to the preprocessing face, the subscriptions data are parsed and stored in an array of **java.lang.String**. Then the native function UpdateClusters(String [][]) is called which updates the structures. After that, the subscription is stored in the *User Log* file maintained in secondary storage, which holds the subscriptions posted and the matched event per subscription, for every subscribed user. This file is also a XML document, and a sample of this file is presented in appendix B. If the last function fails to locate the username of the user posted the subscription in the *User Log* file, a **NoSuchUserException** is thrown and if the *User Log* file is for any reason unreadable, a **LogFileException** is raised. If none of the above, thread finishes normally and suspends.

5.3.3 Event request handling

Each event handling thread must first acquire a read access to the *Prediacate-Cluster* in order to continue processing the event. After that, the event XML document is parsed in an array of **java.lang.String** and passed as argument to the native function String [] CountingAlgorithm(String [][]). The latter is the function implementing the *Counting algorithm* and is written in C++. As expected, this function matches the given event with the subscriptions currently in the *PredicateClusters*, and returns an array of **java.lang.String** containing the subscriptions ids matched (if any). The next step is to browse through the *User Log* file and add the matched event to each statisfied user subscription specified by the subscription id. If the *User Log* file is anyhow unreadable, or if no such user or no such subscription exists in *User Log* file as those specified by the subscription ids, then a **LogFileException**, **NoSuchUserException** or **NoSuchSubscriptionException** is thrown respectively.

5.4 Web application implementation

For system evaluation and demonstration, a web application was build. This application considers the eBay[6] auction site and integrates it with an event

notification mechanism. The application built, applies the notification mechanism on the *eBay cars*[8], which has thousands of auction items (cars). The purpose of this application is to enable users, with the use if a web interface, to post subscriptions and keep track of auction items put up on the eBay site. The application we ipmplemented is composed of two distinct components, the *eBay parser* and the *Web Interface*.

The *eBay parser*, is a set of JAVA classes which can be executed locally or remotely, the purpose of which is to access periodically the eBay auction site and post the eBay events to the *Event Notification* system. Upon *eBay parser* application startup, the *eBay cars* HTML index page is parsed, in which all the brands and models are listed. This information is used in order to build the IE schema and the DTDs over which the events are going to be posted. Then, for every *eBay cars* auction item, a new XML event is formed and posted to the *Event Notification* system.

The web interface enables users to publish subscriptions and events, as well as extend the IE schema to support additional event types, attributes and value domains. Not all functionalities are available to guest users, so a non subscribed user can publish events and extend the schema, while in addition, a subscribed member has his own subscription and matched event repository stored. The interface was build with the use of JSP (JavaServer Pages) and Java Servlets.

In succession, some screenshots of the system with brief comments are presented.

Index page (non-member) In figure 5.5, the index page of the application is shown. The options available to the user are, *publish event*, *add domain*, *add attribute*, *extend schema*, *add event*, *log in* and *subscribe*.

Index page (member) In figure 5.6 is the index page of the application available only to subscribed members. In addition to the non-member index page, the *publish subscripton*, *log off*, *satisfied*, *my subs* options are available. The last two options are links to the matched events per subscription and the posted subscriptions repository page of the user, respectively.

login page This page shown in figure 5.7, is the login page of the system, where the user is prompted for *username* and *password*. On successful login, the user is redirected to the *index page (member)*, else to *index page (non-member)*.

subscribe page As presented in figure 5.8, the user is prompted for *username*, *password* and *password confirmation*. The fields are checked with JavaScripts and only alphanumeric usernames at least 5 digit long are valid.

[8]subset of eBay site

Add domain page As we see in figure 5.9, the user can post a new domain request by choosing the domain value type from a drop-down list and the set of operators associated with that domain from a combo box. If the domain chosen is of *enumerated* type, the user fills a text area with comma separated values specifying the domain value range.

Add attribute page In this page, the user can post a new attribute request by giving the name and the value domain for this attribute . If the attribute name already exists in the IE schema, the user is informed with an error page. The *add attribute* page is presented in figure 5.10.

Extend event schema pages The extension of an event's schema, is a two phase operation and is completed in two pages. In the first page (figure 5.11), the user chooses which event type wish to extend and in the second page (figure 5.12) chooses one or more attributes to add to the schema.

Add event page As shown in figure 5.13, the user can create a new event type by giving a name to the event and specifying one or more attributes to build it's event schema. If an event with the same name already exists in the schema the user is informed with an error page.

Post event Pages The event posting is a two step operation completed in two pages. In the first page, the event type (or types) of the event is specified and in the second page the attribute values are given. The user must fill all mandatory fields. The second page of the operation is illustrated in figure 5.14

Publish subscription pages As pointed out above, this operation is available only to subscribed users. As in event posting, the user first chooses the event type (or types) of the event and then specifies the values and comparison operators of the attributes.The second page of this operation is shown in figure 5.15.

Matched events page In figure 5.16 is shown the page where the matched events per published user subscription are presented.

Published subscriptions page Figure 5.17 illustrates the page where the user can see and manage his subscriptions. The user can delete a subscription in which has no longer interest.

5.5 Client application implementation

The client application Java code, implements a distinct process which connects to the server port and posts events and subscriptions in various rates. Uppon

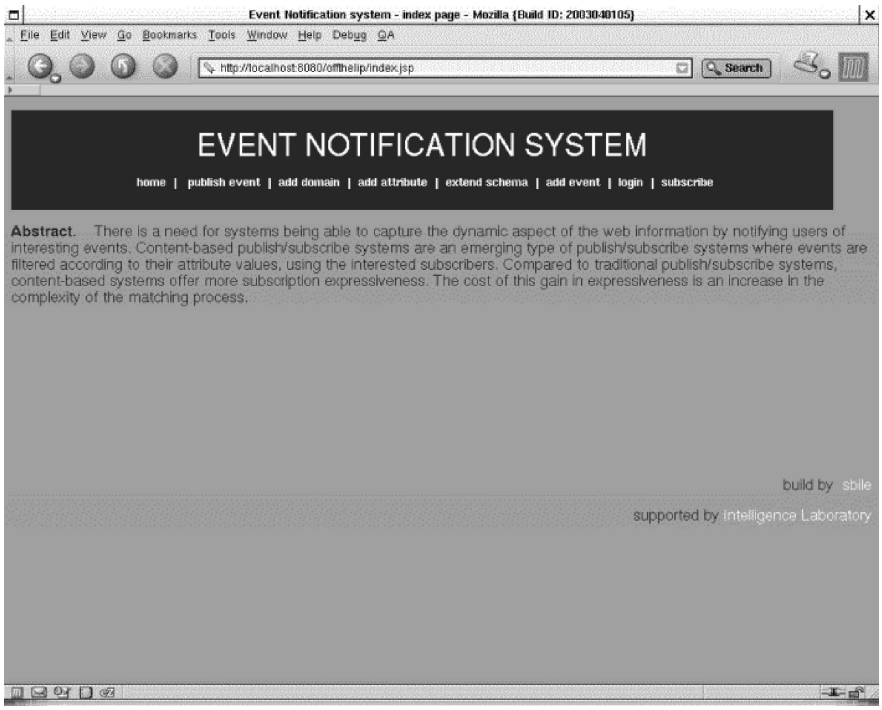

Figure 5.5: Index page (non member)

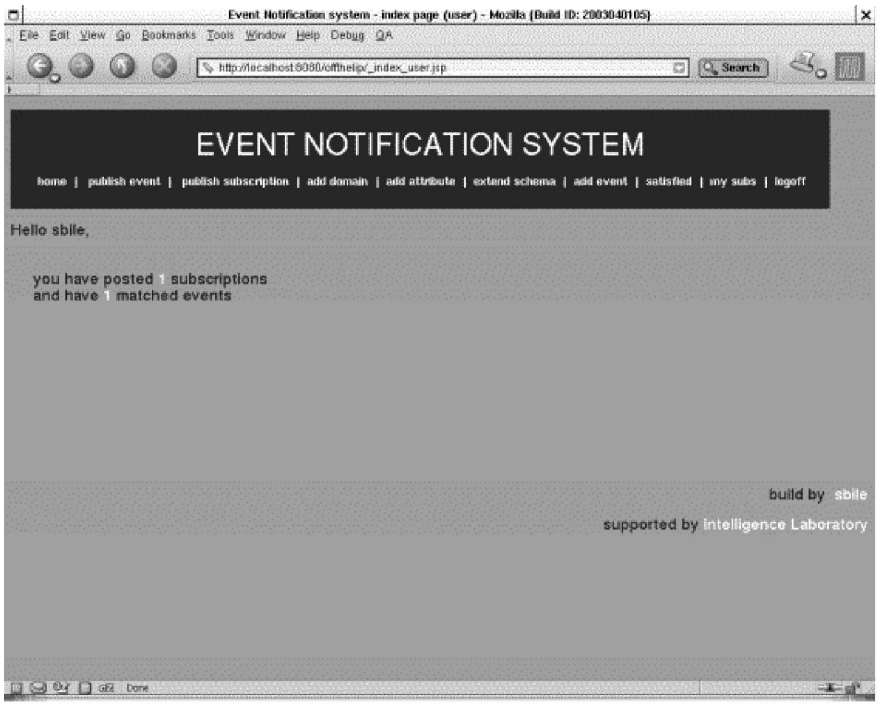

Figure 5.6: Index page (member)

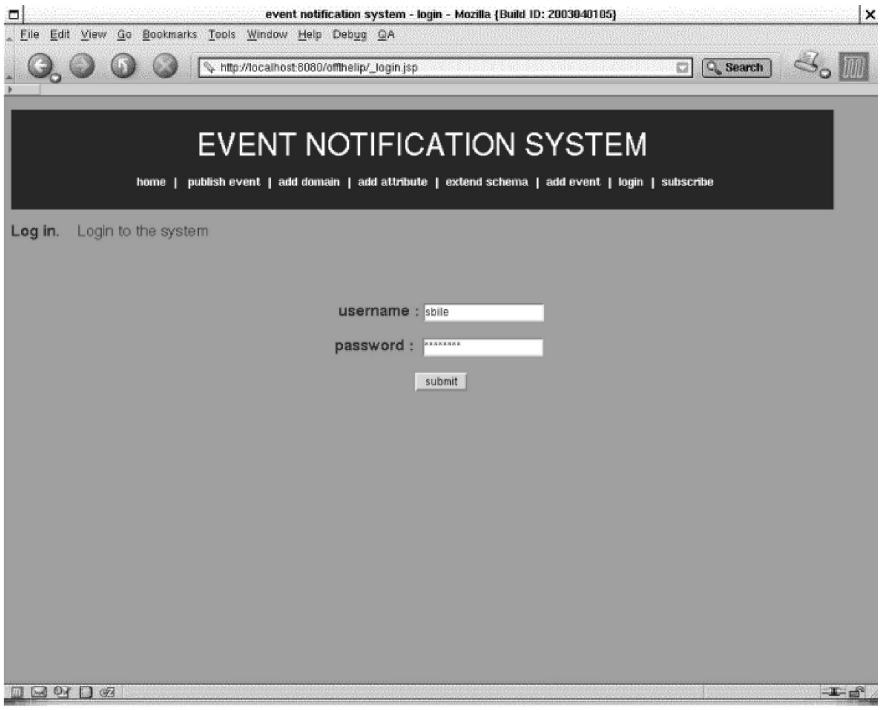

Figure 5.7: Login page

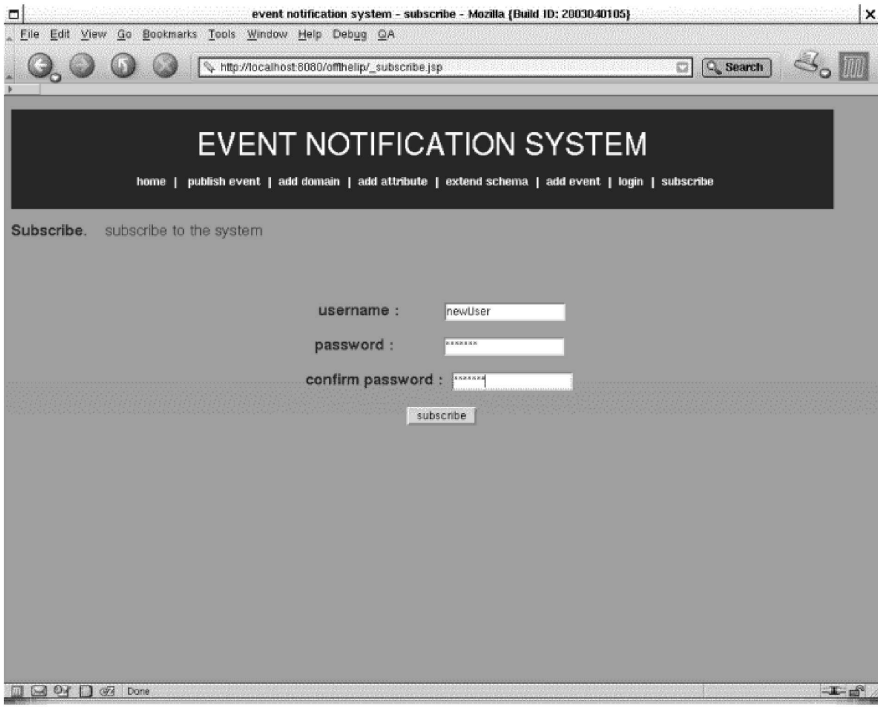

Figure 5.8: Subscribe page

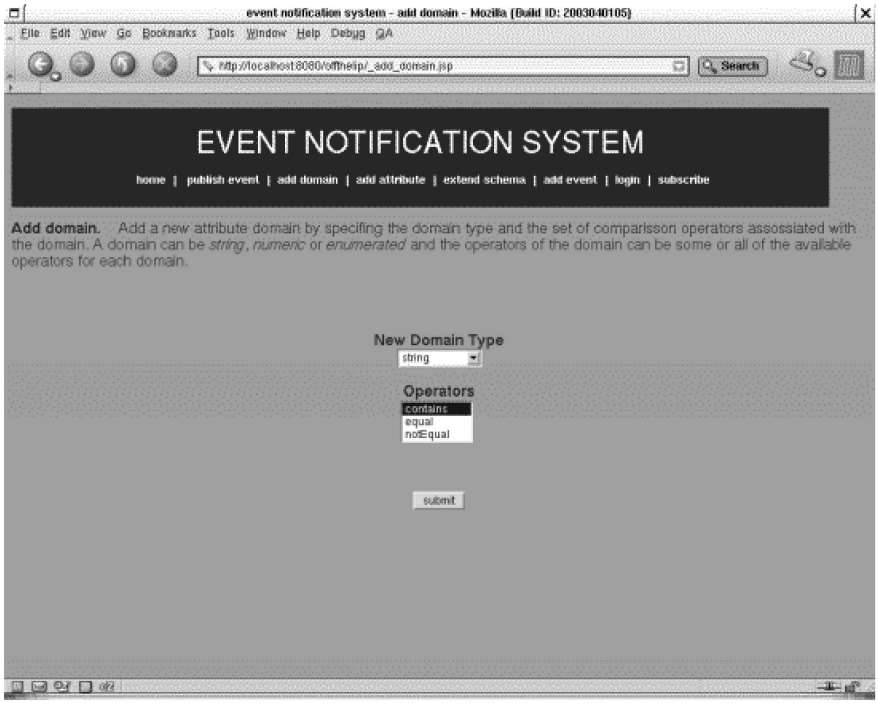

Figure 5.9: Add domain page

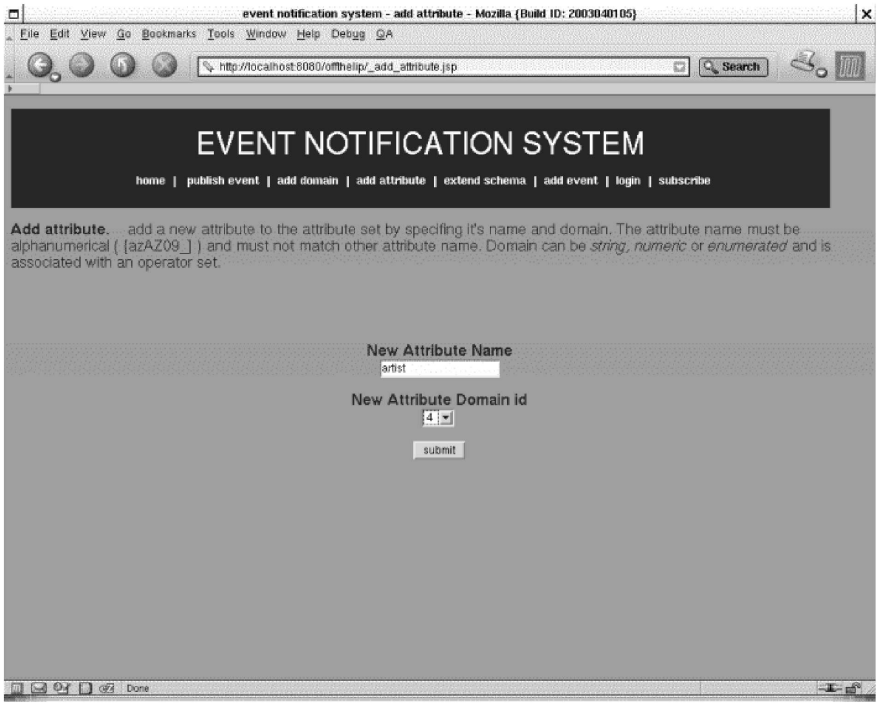

Figure 5.10: Add attribute page

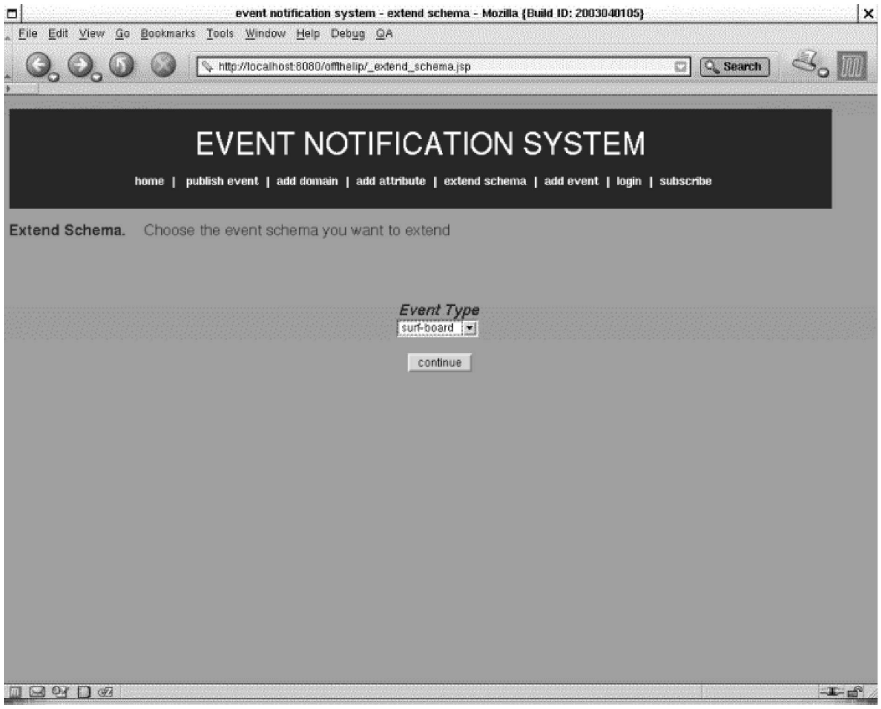

Figure 5.11: Extend event schema page

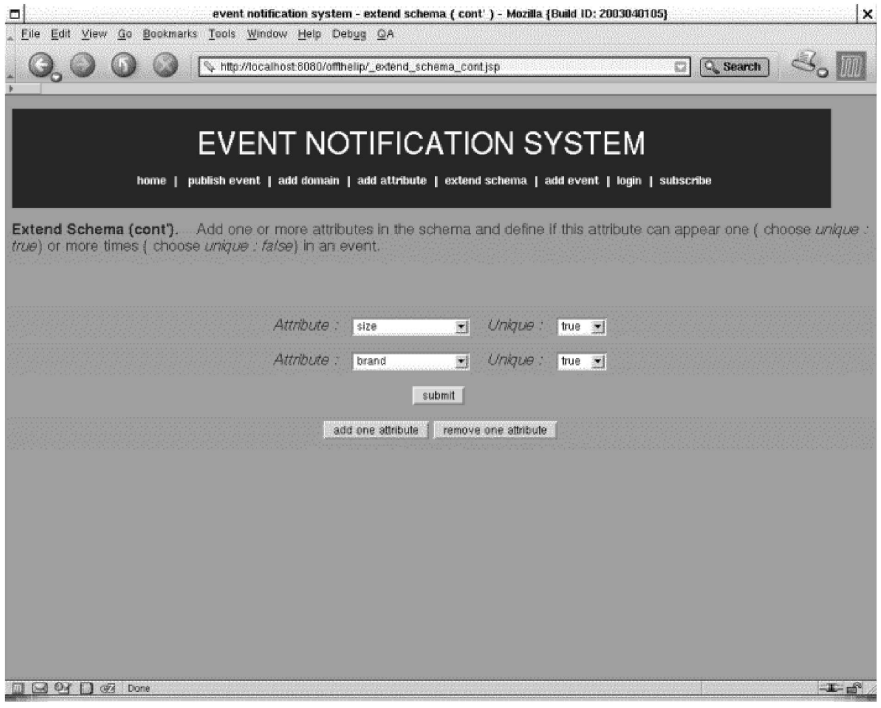

Figure 5.12: Extend event schema page(cont')

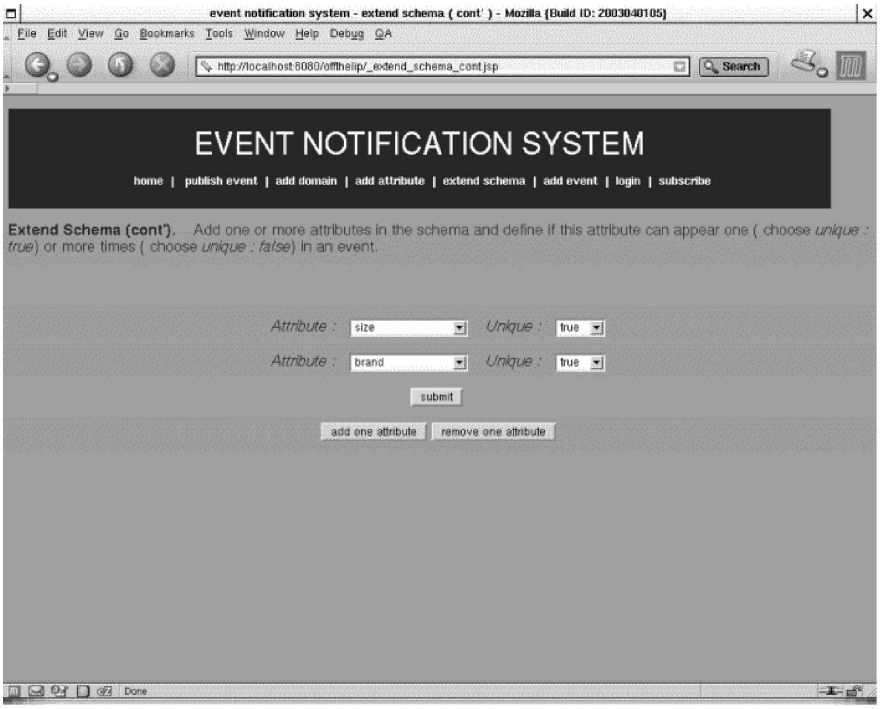

Figure 5.13: Add event page

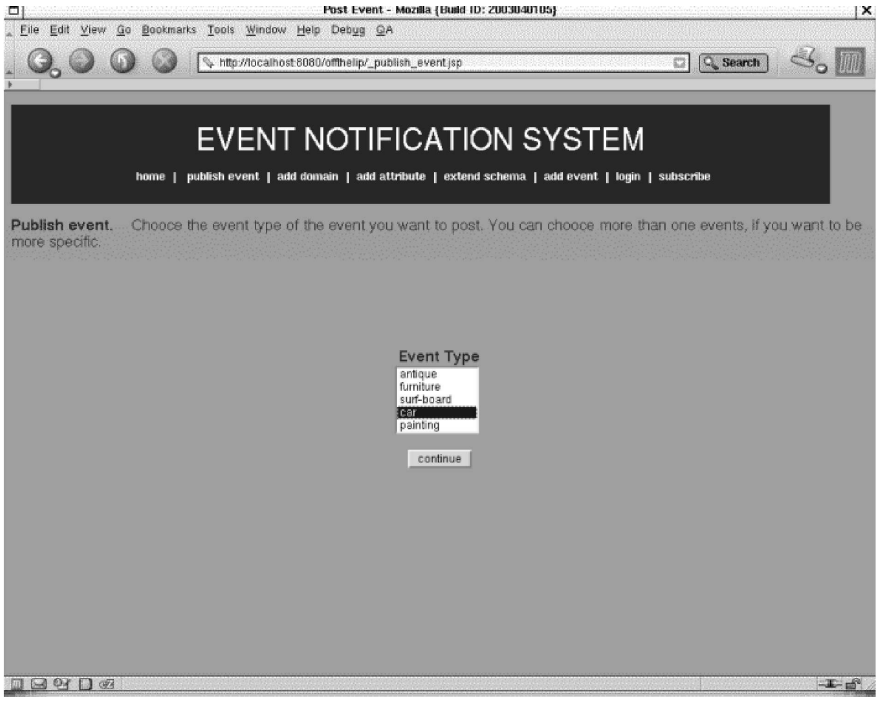

Figure 5.14: Post event page

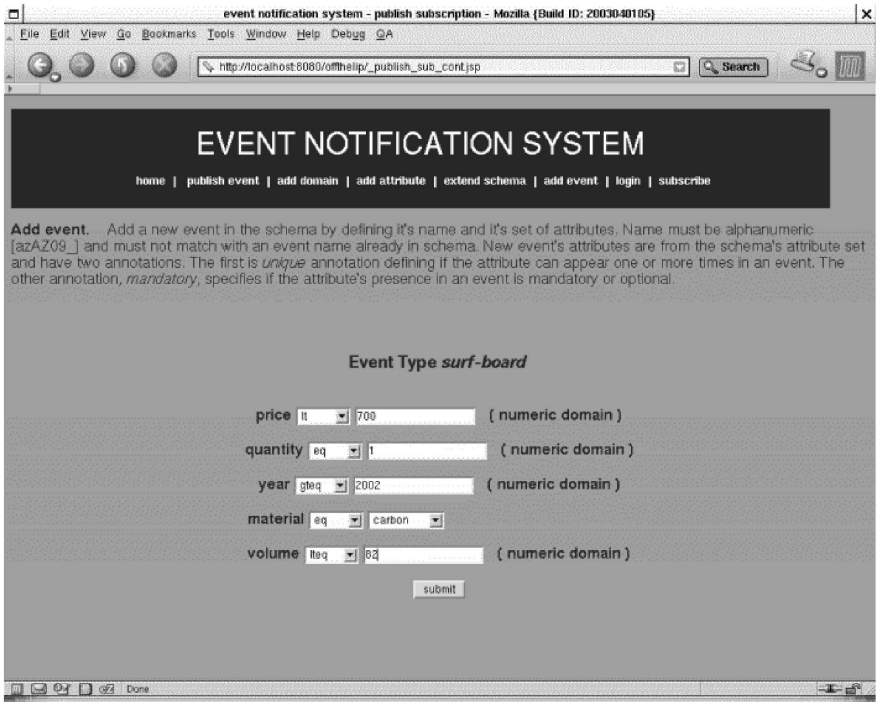

Figure 5.15: Publish subscription page

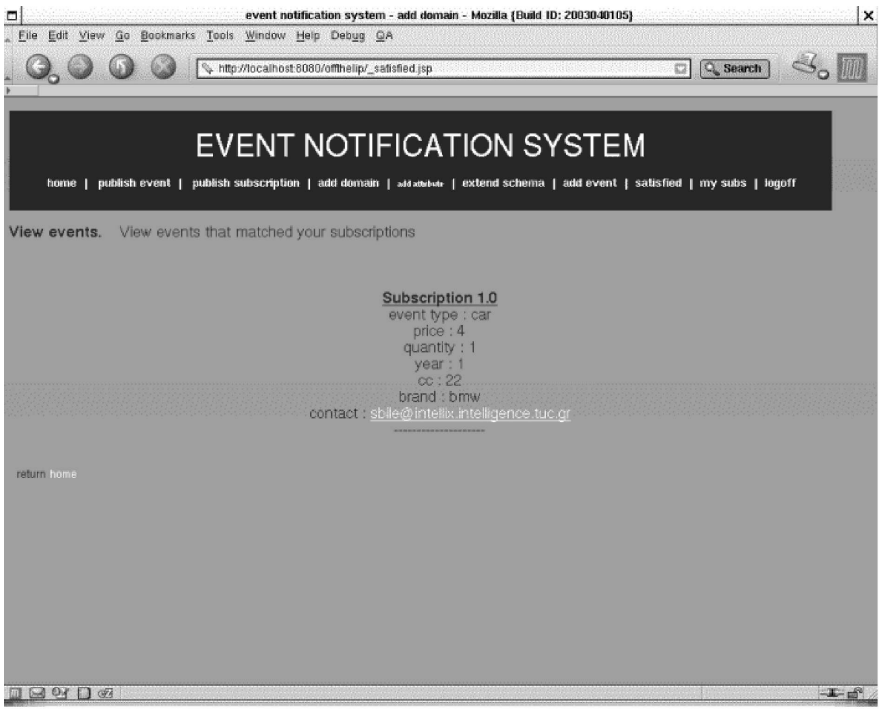

Figure 5.16: Matched event page

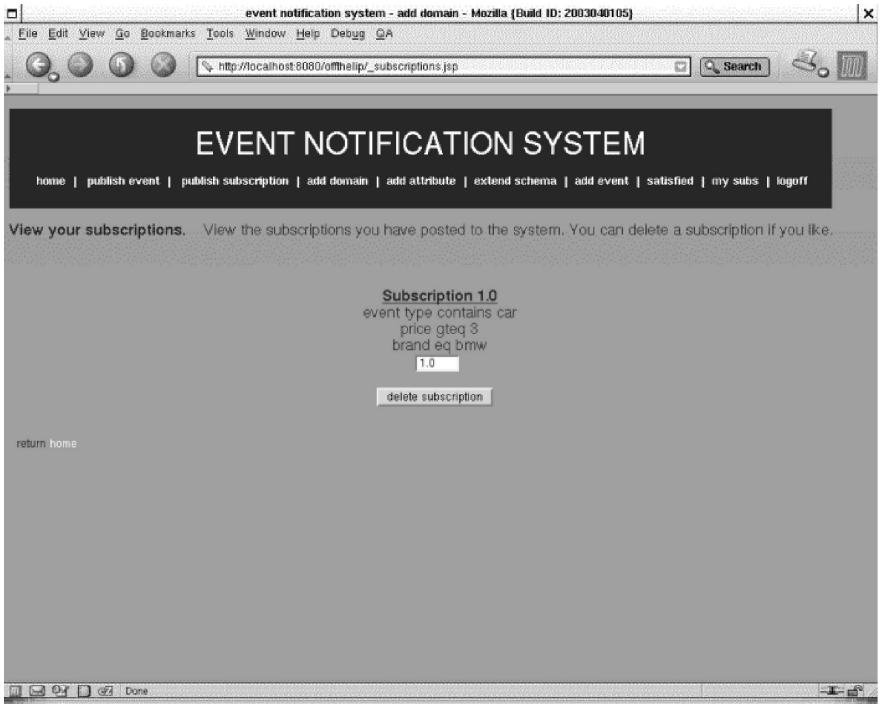

Figure 5.17: Posted subscription page

client startup, a client daemon thread is spawned which in a `while` loop posts request in a ratio of ten events per subscription. In every `while` iteration, a new socket is created, connects with the server process and randomly either a new subscription is created or a new event.

A new subscription XML document is created by calling the `NewRandomSubscription()` function provided by the **thesis.post. Post** class. This function parses the XML IE schema (provided in appendix A) and randomly selects an event type. For each attribute in the event type schema, an operator and value is chosen randomly from its domain. If the domain is of *String* type, there are 150 possible values and only the *contains* operator is available. The attributes with domain type *numeric* can take any value in the field {0,2000}, and can have all the standard comparison operators. Finally, the *enumerated* valued attributes, take random values from a value set specific to each enumerated domain and can appear in *equal* and *not equal* predicates.

Thesis.post.Post class also provides the `NewRandomEvent()` function, which returns a new event XML document. The document is build in the same manner that the subscription documents are build. Namely, after the IE schema is parsed, an event type is randomly chosen. Then for each attribute appearing in that events schema, a random value is given from the attribute value domain.

Chapter 6

Using the API

An event notification mechanism can be applied in a very wide range of applications, the purpose of which are to handle the dynamic aspect of the Web and notify users when interesting events occur. These may vary from stock market updates, weather reports, airplane tickets and other type of information which are highly dynamic and where the searching, quering and retrieving information approach cannot be efficiently applied.

In this context, a demonstration application was built to illustrate the *Event Notification* API usage. We consider the eBay[6] auction site, in which every day a large number of auction items are put up, and present how it can be integrated with an event notification mechanism. We will give in details the procedure to integrate the eBay auction site with an event notification mechanism, which is not exactly the procedure followed in the demonstration application, since access to the eBay database was not available. In order to have the eBay auction items for the demo application, the HTML pages of the site had to by parsed.

6.1 eBay auction site

The eBay[6] auction site is among the most popular auction sites in the Web with thousands of new auction items being put up every day and thousand of users bidding or selling items. eBay sites are currently available in 21 different countries worldwide, each one having distinct databases of users and auction items. In Figure 6.1 we have a quite simplified overview of the eBay site, only illustrating the query and subscribe mechanism of the site. eBay supports only simple string queries over the items which makes finding an item with specific charateristics quite difficult. So, finding an item of interest is not a trivial task considering the huge number of items available and most times more the one queries must be done to narrow down and particularize the results. Moreover, an user may be interested in participating in auctions in distict eBay sites. In this case, the user not only has to repeat queries in each eBay site he is interested in,

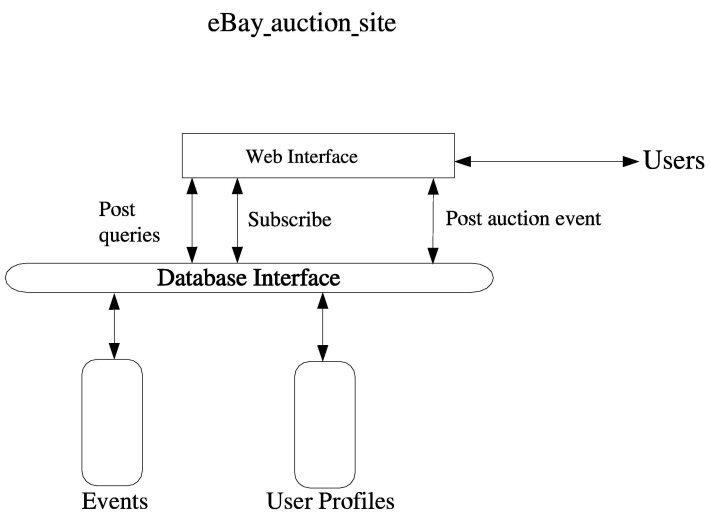

Figure 6.1: eBay auction site

but he has to face with interfaces in different languages.

6.2 eBay auction site(Event Notification integrated)

In figure 6.2 is represented the eBay auction site with the use of the *Event Notification* API. As we see in Figure 6.2, the integrated eBay auction site has the quering functionality which eBay already supports plus the event notification functionality added with the API. This functionality enable users to have *long lasting* queries, which means that an user instead of repeating his queries until an event of interest occurs, he just submits queries in form of subscriptions and he is notified when an auction item matches a subscription.

The integration of the site with the notification mechanism is done in two simple steps, which are the installation of the *Event Notification* server daemon and the extention of the eBay aution site with an interface for event posting, subscribing and extending the IE Schema .

6.2.1 Installing the *Event Notification Server* daemon

One of the reasons that JAVA language is very popular among developers for building Web applications is the code portability. In the application we build, we had to face a dilemma about choosing between code portability and matching algorithm efficiency. The solution was obvious, because the system should be

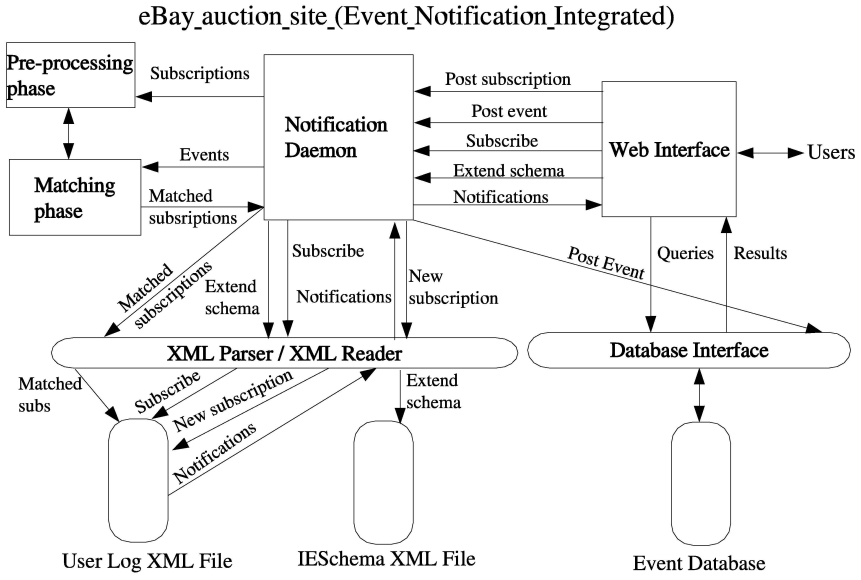

Figure 6.2: eBay auction site using *Event Notifucation* API

efficient enough to cope with high rate of events and large number of subscriptions and this could only be achieved if the matching algorithm was implemented in an efficient language like C++. This fact allows the installation of the *Notification Server* daemon only in Linux environments. Moreover, for the reason that the *Counting Algorithm* is a main memory algorithm the *Event Notification* daemon must be installed and run in a Linux workstation[1] with sufficient CPU speed and RAM, propotional to the number of users expected to use the functionality.

If the above requirements are met, installing and running the daemon, is a very simple task. First, the thesis.jar JAR file must be copied in the installation directory and then the deamon can be excecuted with the java -jar thesis/server/NotificationServerDeamon -*port* -*dir* command. The -jar option enables the execution of a class encapsulated in a jar file, the -*port* option specifies the port the server listens for requests while the -*dir* option specifies the directory in which the *User Log* and *IESchema* XML files will be stored.

6.2.2 Extend eBay site

The eBay auction site must be extended with an comprehensive and user-friendly Web interface, through which the notification functionality will be available to subscribed members. An user, who is subscribed for event notification service,

[1]Not neccessary in the same computer the eBay site is in.

should be able to add a value domain, add an attribute ,add an event type, extend an existing event schema, post an event, publish a subscription, view posted subscriptions and view matched events. Each one of the above operations can be performed from the corresponding Web pages listed below.

add a value domain page In this page, the user should be prompted to choose an value domain from the *string, numeric* and *enumerated* value domains currently supported. Also the set of operators over this domain should be selected from the user. In the case of *enumerated* domains, also a text field should be filled with comma seperated values, specifying the value universe of the enumerated domain. In the end, the public static org.jdom.Document thesis.create.AddDomain(String type, List opers, List enumer) function must be called, which returns an org.jdom.Document object, which represents the XML request. opers and enumer lists are String lists holding the operators and the enumerated values (in case of *enumerated* domain) respectively.

add a new attribute page In this page, the user should be able to add a new attribute by specifying the name and the domain of this attribute. The name should be given in a text field, while the domain should be choosen from a drop-down list, holding all the available domains. Reading the available domains from the IE Schema, is done with the use of the public static List thesis.schema.DomainListWeb() function which returns a list of domains. Finally, the public static org.jdom.Document thesis.create.AddAttribute(String name , String domain) should be called, which returns the request object.

add an event schema page This page should enable the user to add a new event in the IE Schema. This could be done by prompting the user to give a name for the new event and give one or more attributes, from those available in the attribute respository, to make up the new event type. *mandatory* and *unique* notations for each attribute must also be specified by the publisher. The function which reads the attributes from the IE Schema is the public static List thesis.schema.AttrListWeb(). Finally the public static org.jdom.Document AddEventType(String type , List attributes) should be called, taking as parameters the new event name, and a list of thesis.structs.attribute class objects.

extend an event type page In this page, the user should be able to extend the event schema of an existing event type. This could be done by first choosing the event type which he wish to extend and then adding one or more attributes to this event type, not currently in its schema. The XML IE Schema is parsed for the available event types with the public static List thesis.schema.EventListWeb(). In the end, the public static Document ExtendSchema(String event , List attributes) function should be called. attributes is an thesis.structs.attributes class objects list.

post an event page In this page the publisher should be able to choose the event type or types which best describes his event. Then he should fill a form which contains the event attributes (if one event type is chosen), or conjuction of the events attributes (if multiple events are chosen). Each attribute should have a *mandatory* and *unique* notation. The former ranging over values in {mandatory, optional}, which specifies whether a field is mandatory to be filled or not. The latter ranging over values in {unique, multiple}, which determines if the attribute can take one or multiple values. The function which returns the attributes for each event type in a list is public static List thesis.schema.EventListWeb(String eventType). The org.jdom.Document object is created by calling the public static org.jdom.Document PublishEvent(List eventTypes, List attributes), taking two JAVA lists as parameters holding the event type (or types) of the event and the attributes of the event.

publish a subscription In this page the operation of subscription posting must take place. During this procedure the user should be first promted to signify the event type (or event types) in which is interested in posting the subscription. Then he should fill a form which contains the event attributes (if one event type is chosen), or conjuction of the events attributes (if multiple events are chosen). A subscription should be a set of triples of the form (*attribute name, operator, attribute value*). A form should be valid if at least one such triple is defined. In every subscription posted by a user, the *User Log* file will be updated which holds all the subscriptions published by the subscribers. To form an org.jdom.Document from the information collected, the public static org.jdom.Document PublishSub(List eventTypes, List subAttributes) should be engaged. This functions parameters are two JAVA lists, the first holding the event type (or types) of the subscription, while the second holds the thesis.structs.subAttributes class objects.

view posted subscription page In this page, all the subsctipions posted by the user should be presented. The subscriptions posted by each user are kept in the *User Log* XML file and can be accessed by the public static java.util.List thesis.schema.subscriptions(String user) function which returns a list of thesis.structs.triple class objects. Moreover, the user should able to delete a subscription. This is done with the public static void thesis.schema.seleteSub(String subID)

view matched events page In this page, all the events matched by user subscriptions must be listed. This is done with the public static List thesis.schema.matchedEvents(String user) function which returns a list of thesis.structs.event class objects.

After every request creation, the public static void thesis.post.postRequest(org.jdom.Document request) function should be called, which is responsible for

forming an XML request from the org.jdom.Document object and posting it to the *Event Notification* daemon.More information about the classes and functions of the API can be accessed with the javadoc tool.

Chapter 7

Counting Algorithm Implementation

The *counting algorithm* corresponds to a set of C++ classes and functions compiled and linked in the shared library `libCounting.so`. This library is linked dynamically with the *NotificationServer* code on server startup through Java Native Interface (JNI), which provides a convenient and easy way for integrating Java with native code. This is very useful for time critical applications because allows part of code (like algorithms) that the excecution time is an important factor, to be implemented in more efficient languages (like C++). The declaration of the native functions in the Java code is similar to the plain Java functions except the additional `native` keyword preceding each method[1].

The *Counting algorithm* has four distinct phases, the *pre-processing*, the *matching*, the *hit-applying* and the *counting* phase. The first corresponds to the execution of the native function UpdateClusters(String [][]) which is engaged when a new subscription arrives, while the other three correspond to the execution of the native function String [] CountingAlgorithm(String [][]) which is called after a new event reaches the server.

7.1 Pre-processing phase implementation

In this phase two in-memory structures are kept and updated by the UpdateClusters(String [][]) function. The first structure called *PredicateClusters*, as shown in Figure 7.2, consists of six hash tables of vectors, the *EqPred*, *LessPred*, *LessEqPred*, *GreaterPred*, *GreaterEqPred* and the *ContainsPred* hash table, which hold the equality, less, less equal, greater, greater equal and contains predicates respectively. Each predicate is represented by an Predicate class entry in *PredicateClusters* structure. The *Predicate* class has four fields to store the name, operator, value and domain of the predicate. The *contains* predicates have an

[1] no function body is given in this context

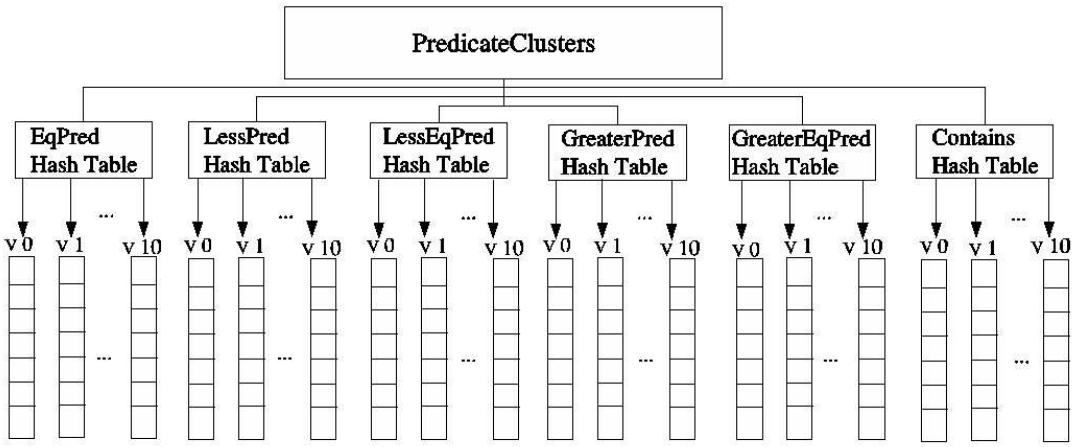

Figure 7.1: Predicate to subscription association table

additional field which is a vector holding comma, space and dot separated tokens of their value. This is useful when the value of a predicate is of *string* type. The usage of this vector becomes clear below.

The second structure called *PredToSub*, as illustrated in Figure 7.1, is an association table which holds the mapping between the predicates and the subscriptions they appear in. It also consists of six hash tables of vectors, the *EqPred*, *LessPred*, *LessEqPred*, *GreaterPred*, *GreaterEqPred* and the *ContainsPred* hash table.

As described in 5 chapter, when a new subscription comes, it is decomposed in predicates and is sent to the native function UpdateClusters(String [][]) as an array of *attribute, operator, value* triples. Then, each predicate is hashed with the hash code returned from the int HashFunction(char *name) function, to the appropriate hash table of the *PredToSub* structure, according to its operator. If an identical predicate entry is already in that vector, the subscription in which the predicate was found is added to the list of the subscriptions the predicate was found so far, else a new predicate entry is made in the vector containing the predicate name, value and the subscription in which it appeared. In the first case, where a predicate is already in the *PredToSub* structure, no new entry has to be done to the *PredicateClusters*, because each predicate is stored only once. In the second case however, the predicate is hashed to the appropriate hash table of the *PredicateClusters* structure and a new entry is added to the corresponding hash table vector.

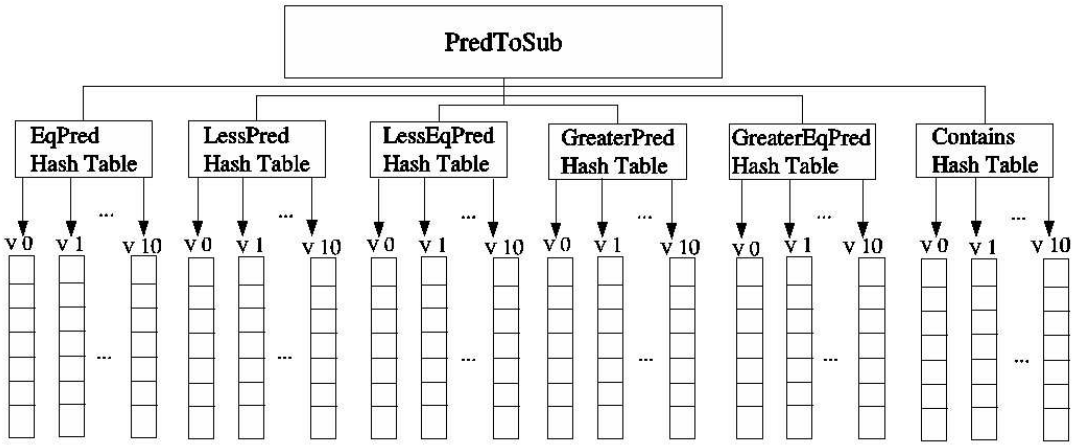

Figure 7.2: Predicate Clusters representation

7.2 Matching phase implementation

When a new event reaches the server a sequence of actions occur in order to be processed. As already discused in previous chapter, after the XML event is parsed and decomposed in predicates, is send to the native function CountingAlgorithm(String [][]) as an array of *attribute, value* pairs. Then, each *pair* is given as input to the equalpredmatch(pair p), lesspredmatch(pair p), lesseqpredmatch(pair p), greaterpredmatch(pair p), greatereqpredmatch(pair p) and containspredmatch(pair p) functions, which are responsible for matching each pair with the equality (and non equality), less, less equal, greater, greater equal and contains predicates in the *PredicateClusters* respectively.

The equalpredmatch(pair p) function searches, for each event's *pair p*, the predicate with the same value as the *p* value in the *equality* cluster associated to the attribute hash code. Each cluster of equality predicates is kept suitably ordered by adding new elements with a binary insertion algorithm. If a predicate with the same value is found is inserted in a vector called *satisfiedPredicates*.

The lesseqpredmatch(pair p) searches, for each event's *pair p*, the predicate with the greatest value less than or equal to the *p* value in the *less than equal* cluster associated to the attribute hash code. After having found this predicate, we know that all predicates that are placed before (while predicates are stored in increasing order) are also satisfied by the event. We use a binary search algorithm to find the predicate with the greatest value less than or equal to the *p* value. The greatereqpredmatch(pair p) function is similar to lesseqpredmatch(pair p). The predicates satisfied by the *p* value are stored in the *satisfiedPredicate* vector.

The lesspredmatch(pair p) function searches, for each event's *pair p*, the pred-

icate with the greatest value less but not equal to the p value. Since predicates are kept ordered with a binary algorithm, all the predicates below the one found are also satisfied. The greaterpredmatch(pair p) algorithm works in a similar way. If satisfied predicates found by the p value, they are added to the *satisfiedPredicate* vector.

The containspredmatch(pair p) function works a little bit different from the other functions. As illustrated above, each *Predicate* class entry of the *PredicateClusters* structure, has a vector field which holds comma, space and dot separated tokens of the predicate value. The matching of each events *pair p* with a *contains* predicate, is not done by comparing the two values, but by searching for each token in the *Predicate* tokens vector for a matching pattern in the p value. If one is found then the predicates is satisfied by p and is stored in the *satisfiedPredicate* vector.

The matching of each *pair* to the corresponding hash table vector, is done by comparing the hash code of the pair value, with the hash code of each predicate value in the *PredicateClusters*. In comparisons, we choose the hash code of the value instead of the value itself, because the hash code permits numeric operations regardless of the value type[2].

7.3 Hit applying phase implementation

For the *hit applying* phase, a new structure needs to be used. This structure, is a vector of subscription classes and is called *satisfiedSubscription*. The subscription class has three fields, the *subscriptionID* field, which holds the subscription unique id, the *NumberOfPredicates*, which holds the number of predicates found in the subscription and the *HitCount* field, which is initialized to zero and its purpose is discussed in details below.

In this phase, we need to count the number of predicates satisfied for each subscription in the *matching* phase of the algorithm. This is implemented by applying a *hit* to every subscription having a satisfied predicate. So, for each predicate in *SatisfiedPredicates*, a binary search is performed in the *PredToSub* in order to find the subscriptions in which each predicate exists. For each subscription found, either a new entry is added in the *satisfiedSubscriptions* vector and the *HitCount* is set to one, or, if there is already an entry in the *satisfiedSubscriptions* with the same *subscriptionID*, then the *HitCount* of that entry is increased by one.

[2] numeric, enumerated, string.

7.4 Counting phase implementation

In the last phase of the algorithm, we just traverse through the *satisfiedSubscriptions* vector and compare the values of the *NumberOfPredicates* and *HitCount* fields of each entry. If the two numbers are equal, then the event has satisfied the specific subscription, as all the predicates of the specific subscription are matched by the event.

The String [] CountingAlgorithm(String [][]) native function returns an array of subscription ids of the subscriptions satisfied by the event processed.

7.5 Algorithm snap-shots

Suppose a senario of three subscriptions s_1, s_2 and s_3 being posted to the system, where

s_1 = {(a>6), (b=6), (c<12)},

s_2 = {(a>6), (d=7},

s_3 = {(d=7, (e<3)}

The subscriptions are then processed by the *counting* algorithm. In the *pre-processing* phase of the algorithm the subscriptions are decomposed in their elementary *predicates*. So we have :

$s_1 \Longrightarrow p_1$: (a>6), p_2 : (b=6), p_3 : (c<12),

$s_2 \Longrightarrow p_1$: (a>6), p_4 : (d=7),

$s_3 \Longrightarrow p_4$: (d=7), p_5 : (e<3)

Identical predicates share the same number.

In Figure 7.3 we have a snap-shot of the *PredToSub* struct after predicates insertion. Each predicate is inserted only once and is associated with the subscriptions it appears in. Each predicate is inserted in the appropriate cluster with respect to the operator it appears in and hashed to a vector according to its name hash code. Vectors are kept sorted with a binary insertion algorithm.

Figure 7.4 presents *Predicates* struct after predicate insertion. Each predicate is inserted to a cluster specific to its comparison operator and then hashed to a vector according to its operator. Predicates are inserted to vectors with a binary insertion algorithm. In this point, the *pre-processing* phase is complete.

Lets now see how a new event e = {(c,8), (e,0), (d,9)} is handled by the *counting* algorithm. Event e is decomposed to *pairs (c,8), (e, 0)* and *(d,9).*

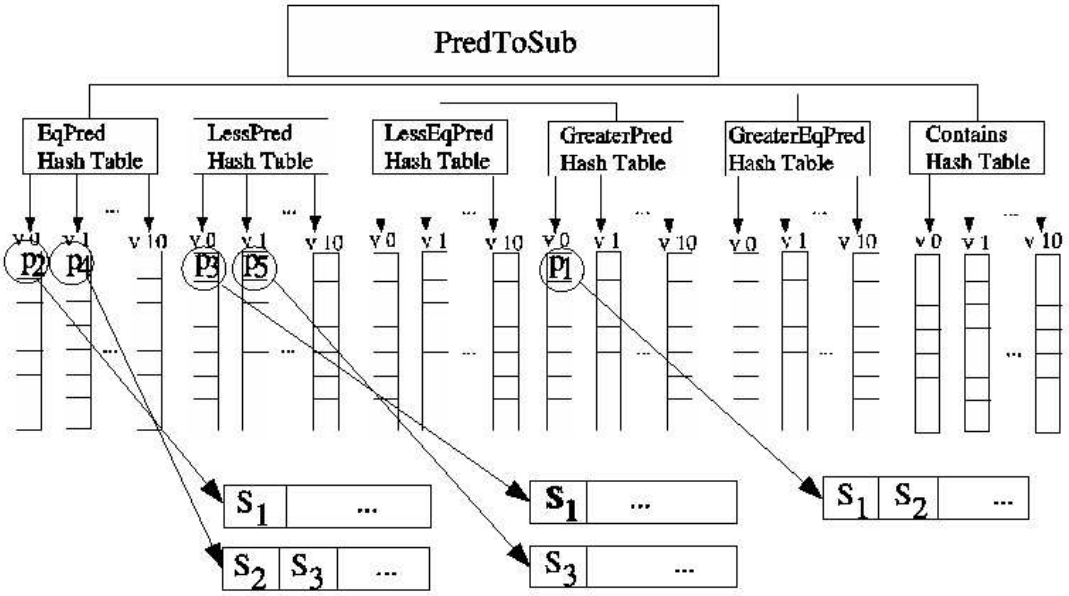

Figure 7.3: PredToSub clusters struct snap-shot

Then each pair of e is matched against predicates in *Clusters* struct, as described in 7.2. To repeat briefly, each event pair is hashed to the appropriate cluster vector according to its operator and name hash code. Then, predicates satisfied by the event pair are found with a method based on binary search and stored in *SatisfiedPredicate* vector. Snap-shot of *SatisfiedPredicate* after all events pair evaluation is shown in Figure 7.5.

In this point, from the set of satisfied predicates we must specify the satisfied subscriptions. For this phase, one additional structure is needed called *Satisfied-Subscriptions*. For every predicate in *SatisfiedPredicate* vector, we run through *PredToSub* structure and find the subscriptions in which the satisfied predicate appeared[3]. For each subscription found, depending on if there is already a subscription entry in *SatisfiedSubscriptions* vector with the same id or not, either a new entry is made to the *SatisfiedSubscriptions* vector and the entry variable `Hits_count` is initialized to one or the `Hits_count` variable of the subscription entry already in the *SatisfiedSubscription* vector is increased by one. After this process the *SatisfiedSubscriptions* vector is as illustrated in Figure 7.7.

[3]Figure 7.6 illustrates the data structs of *PredToSub* structure.

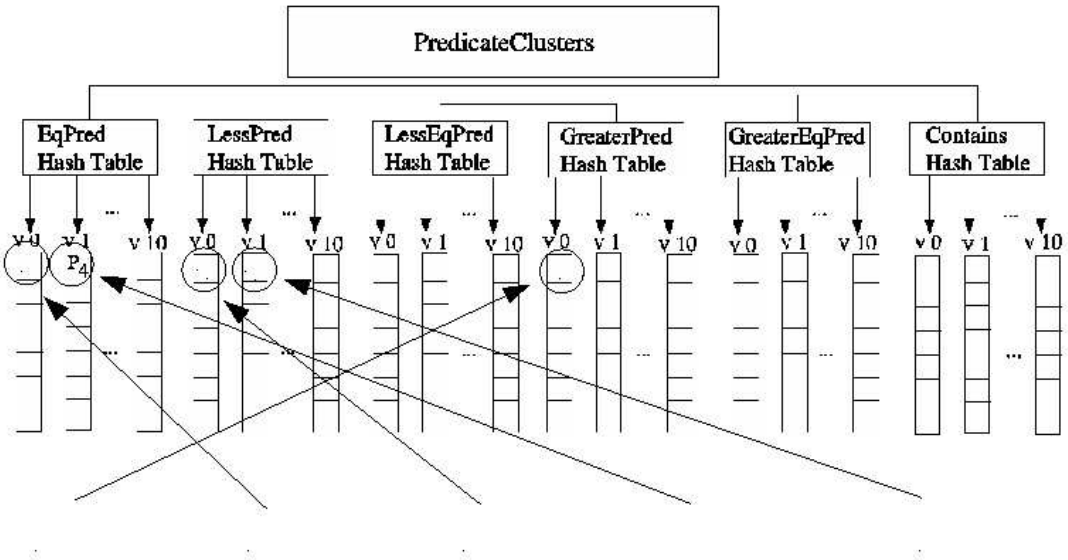

Figure 7.4: Predicates clusters struct snap-shot

Finally, *SatisfiedSubscriptions* vector is traversed and every subscription found having the `Hits_count` and `#_of_predicates` variables equal is returned as satisfied, because all subscription predicates are satisfied.

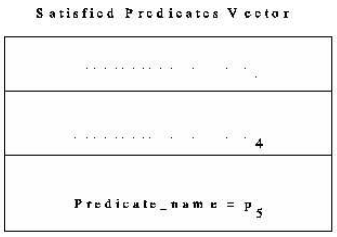

Figure 7.5: *SatisfiedPredicate* vector snap-shot

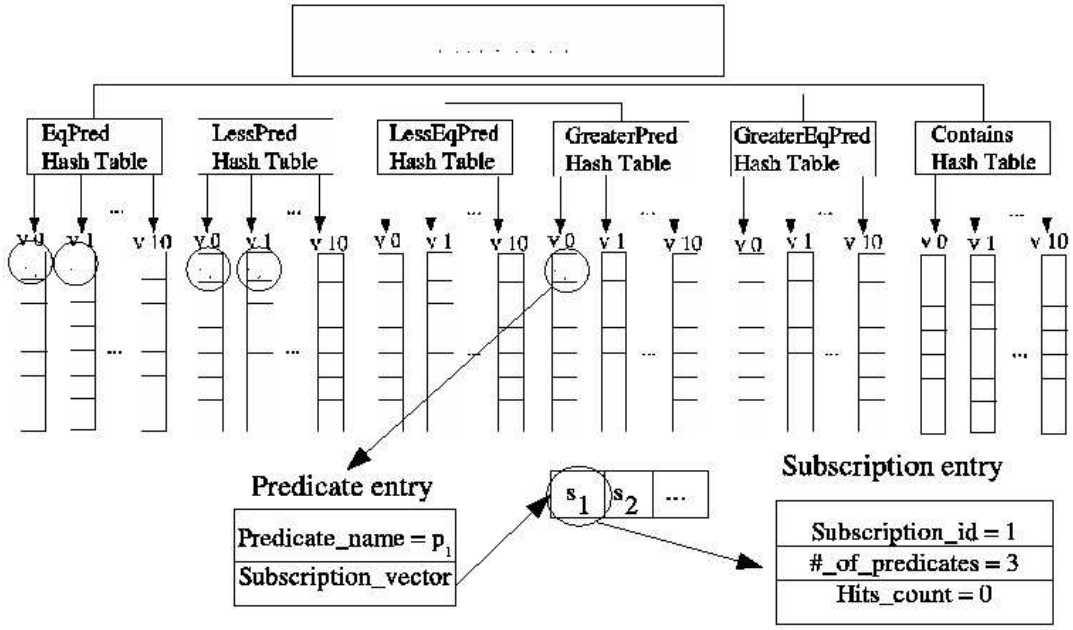

Figure 7.6: *PredToSub* data structs

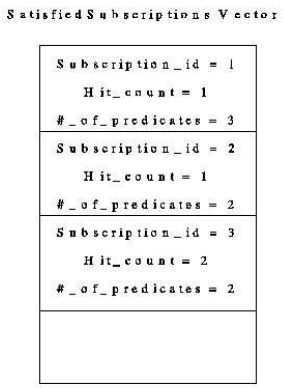

Figure 7.7: *SatisfiedSubscriptions* vector snap-shot

Chapter 8

Running Simulation

In this chapter are shown the experimental results of performance tests of the matching algorithm with a variety of work loads. The performance results discussed bellow are performed on a single-CPU Pentium workstation with a i686 CPU at 550MHZ and 256MB RAM running Linux. The generation of events and the excecution 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 in a rate of 20 events/sec. This one excecutes the matching algorithm and sends the result back to the first process. The result is represented by an array of matched subscription ids per event. Both processes correspond to the execution of JAVA programs.

8.1 Random subscription generation

In what concerns the generation of subscriptions, they are randomly generated assuming the IE schema of Appendix A. So, every subscription generated, has an equal probability in appearing in each one of the five event types of the sample IE schema, used for the simulation. Every events schema, consists of four mandatory and one optional attributes. A subscription belongs in one event type and all the event type's attributes are present in each subscription. Each subscription predicate can take value and operator uniformly distributed over their value domain and operator set respectively.

The values with domain type *string*, have 150 possible values and can appear only in *contains* predicates. The values with domain type *numeric* can take any value in the field {0,2000}, and can have all the standard comparison operators. Finally, the *enumerated* valued attributes, take random values from a value set specific to each enumerated domain and can appear in *equal* and *not equal* predicates.

8.2 Random event generation

The Random event Generator represents a distinct Java program which runs localy and its purpose is to post events and subscriptions to the Notification server, under various work loads and in various rates. So, with the use of this client application we can evaluate the matching performance of the algorithm for various subscription numbers, and of course to evaluate the algorithm correctness.

Performance evaluation In order to evaluate algorithm's performance, the client must be able to send subscriptions and events in a high rate. Indeed, the client sends subscriptions and events in a rate of 20 per second. The client application generates random subscriptions with respect to a sample schema of Appendix A.

Correctness evaluation The client has a second mode of operation, in which the user can see the results of the matching process in a terminal. The purpose of this operation is basically for debugging and used during the algorithm development stage.

Every event generated, has an event type randomly chosen between the five available event types of the sample IE schema of Appendix A. Each event can appear in any of the event types of the sample IE schema [1]. Every events schema, consists of four mandatory and one optional attribute, which take values uniformly distributed over their value domain.

8.3 Evaluate results

The processing of an event is the sum of the excecution time of the matching algorithm plus the communication time between the *Java* and *C++* program. In the following figures the communication time is not taken into account, and only the excecution time of the *Counting Algorithm*[2] is evaluated.

8.3.1 Evaluate event matching

As shown in Figure 8.1, the *Counting algorithm* performance is close to linear, which makes it quite predictable and efficient for environments with high event rates and large number of subscriptions. Moreover, *Counting Algorithm* is a predicate based algorithm and its efficiency depends on the high redundancy of predicates, because (unlike the *naive* approach) does not reevaluate redundant predicates. So, taking into account that a number of subscriptions very large compared to the size of attribute domains incurs in a high predicate redudancy

[1] Just like in random subscription creation, discussed above.
[2] The *C++* program.

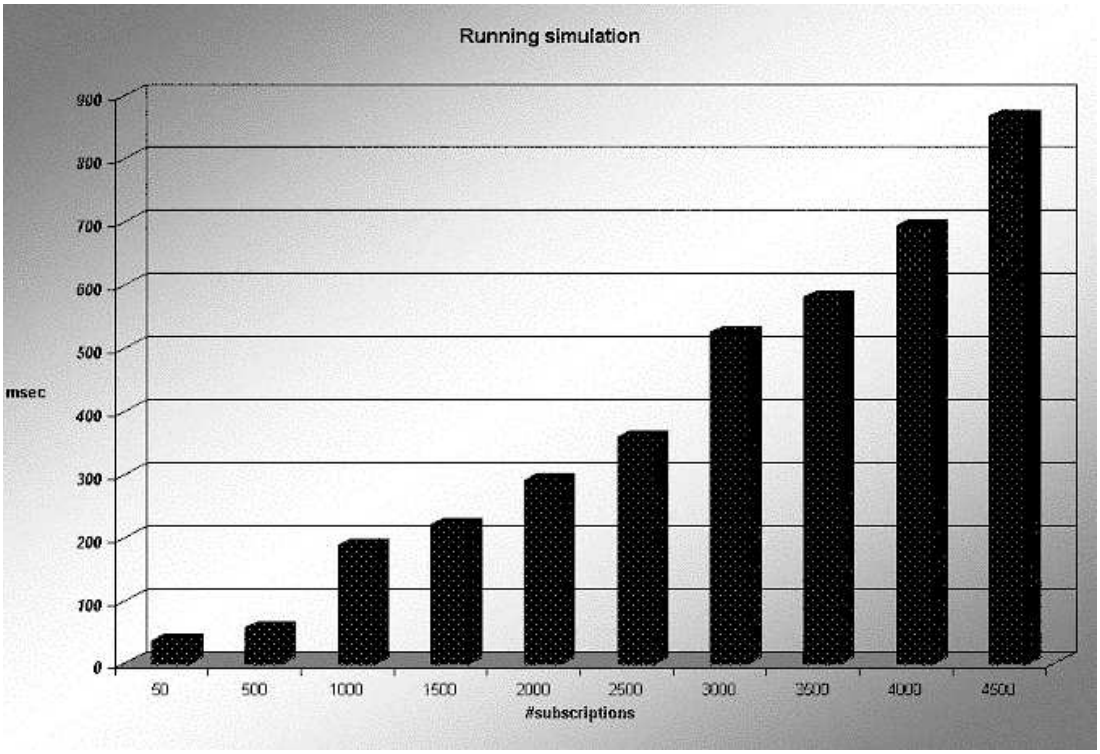

Figure 8.1: Event processing results

and the fact that this is true in the web context where many enumerated values are manipulated (e.g. categories, names, locations), we conclude that *Counting Algorithm* is an algorithm suitable for the Web.

8.3.2 Evaluate new subscription processing

For the reason that a new suscription processing involves writing to the predicates hash table, neither any other new subscription can be processed concurrently, nor can an event have read access to the predicate table in order to be matched, for as long the subscription is updating the table. So, for the overall performance of the algorithm, it is critical that the processing of a new subscription is efficiently implemented.

As shown in Figure 8.2, the predicate table updating with a new subscription is very fast. It is very interesting to note that the insertion time of a new subscription is not always proporsional to the subscriptions already in the table. That is fairly reasonalbe, considering the redundancy of the predicates. So, when a subscription insertion time is low, we have to assume that most of the predicates in the subscription were already in the table.

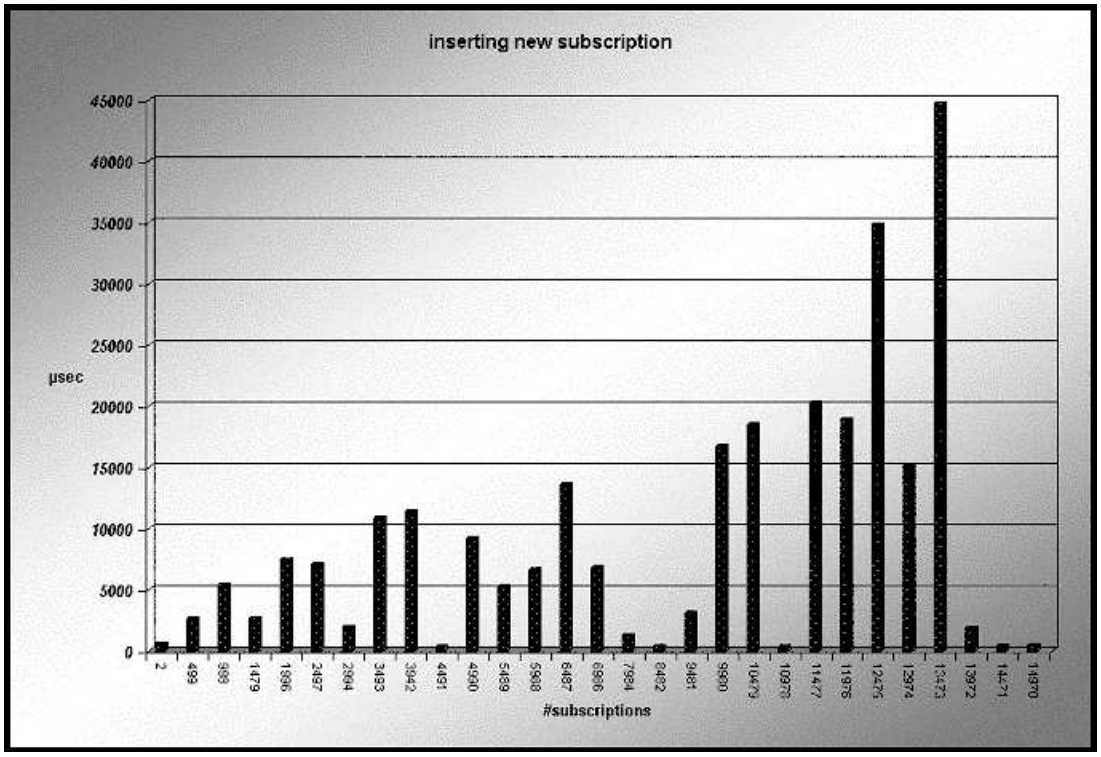

Figure 8.2: Subscription processing results

Chapter 9

Conclusions

The system we present in this work is an implementation of the system proposed in [22], called *Le Subscribe*. The main features of this system are an event model which permits easy representation and integration of the information published on the Web; a subscription language supporting the most usual queries while being quite expressive; A matching algorithm which integrates well with high rate of events and a large number of subscriptions; An easy to use API enabling easy integration of existing systems with an event notification mechanism.

Of course, the core feature of the system is the *Counting algorithm*. The charateristics which makes it very efficient for the Web context are that it is a main memory algorithm and predicate based. The first means that no access to secondary storage is necessary, while an in-memory structure is kept holding the subscriptions predicates while the events are proccessed on the fly. The second charasteristic means that the algorithm is pure predicate based, applying a global optimization strategy by exploiting the predicate redudancy.

Moreover, the XML event and subscription matching against the publication schema is performed upon XML parsing with the use of DTD files. The DTD files are automatically rearranged on every IE schema modification operation.

All these features are available through a simple API which enables easy integration of a system to use *Event Notification* mechanism. The API is fairly lightweigth, multithreaded and the time critical parts of the code are implemented in native functions for efficiency.

Much work remains to be done in this context. Currently the subscription language only supports subscriptions which are conjuctions of elementary predicates. The language can be enriched with disjunctive queries. Moreover, in addition with *string*, *numeric* and *enumerated* domains, *hierarchical* domains could be supported. The *hierarchical* domain is an enumerated domain whose elements are organized according to a hierarchy and is useful to depict categories and subcategories[1]. Of course, a distributed version of the system can be easily

[1]This feature is supported in Le Subscribe

developed, based on XML communication between the decentralized servers.

Bibliography

[1] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. Strom, and D. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. In *19th International Conference on Distributed Computing Systems (19th ICDCS'99)*, Austin, Texas, May 1999. IEEE.

[2] D. Bertsekas and R. Gallager. *Data Networks*. Prentice Hall, 1987.

[3] A. Carzaniga, D.S. Rosenblum, and A. L. Wolf. Achieving scalability and expressiveness in an internet-scale event notification service. In *Proceedings of the 19th ACM Symposium on Principles of Distributed Computing (PODC'2000)*, pages 219–227, 2000.

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

[5] Arvola Chan. Transactional publish/subscribe: The proactive multicast of database changes. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD-98)*, volume 27,2 of *ACM SIGMOD Record*, pages 521–521, New York, June 1–4 1998. ACM Press.

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

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

[8] F. Fabret, F. Llirbat, J. Pereira, and D. Shasha. Efficient matching for content-based publish/subscribe systems. Technical report, INRIA, 2000. http://www-caravel.inria.fr/pereira/matching.ps.

[9] Françoise Fabret, H. Arno Jacobsen, François Llirbat, João Pereira, Kenneth A. Ross, and Dennis Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 30(2):115–126, 2001.

[10] A. Galardo-Antolin, A. Navia-Vasquez, H.Y. Molina-Bulla, A.B. Rodriquez-Gonzalez, F.J. Valvarde-Albacete, A.R. Figueiras-Vidal, T. Koutris,

A. Xiruhaki, and M. Koubarakis. I-Gaia: an Information Processing Layer for the DIET Platform . In *Proceedings of the 1st International Joint Conference on Autonomous Agents & Multiagent Systems (AAMAS 2002)*, pages 1272–1279, July 15–19 2002.

[11] R. Gruber, B. Krishnamurthy, and E. Panagos. The architecture of the ready event notification service, 1999.

[12] E. N. Hanson. A predicate matching algorithm for database rule systems,. Technical Report WSU-CS-90-01, Washington State University, 1990.

[13] Cefn Hoile, Fang Wang, Erwin Bonsma, and Paul Marrow. Core specification and experiments in DIET: a decentralised ecosystem-inspired mobile agent system. In *Proceedings of the 1st International Joint Conference on Autonomous Agents & Multiagent Systems (AAMAS 2002)*, pages 623–630, July 15–19 2002.

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

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

[16] M. Koubarakis, T. Koutris, P. Raftopoulou, and C. Tryfonopoulos. Information Alert in Distributed Digital Libraries: The Models, Languages and Architecture of DIAS. In *Proceedings of the 6th European Conference on Research and Advanced Technology for Digital Libraries (ECDL 2002)*, volume 2458 of *Lecture Notes in Computer Science*, pages 527–542, September 2002.

[17] P. Marrow, M. Koubarakis, R.H. van Lengen, F. Valverde-Albacete, E. Bonsma, J. Cid-Suerio, A.R. Figueiras-Vidal, A. Gallardo-Antolin, C. Hoile, T. Koutris, H. Molina-Bulla, A. Navia-Vazquez, P. Raftopoulou, N. Skarmeas, C. Tryfonopoulos, F. Wang, and C. Xiruhaki. Agents in Decentralised Information Ecosystems: The DIET Approach. In M. Schroeder and K. Stathis, editors, *Proceedings of the AISB'01 Symposium on Information Agents for Electronic Commerce, AISB'01 Convention*, pages 109–117, University of York, United Kingdom, March 2001.

[18] H. McLaughlin, J. XML, and w JDOM.

[19] D. Megginson. "SAX 2.0: The Simple API for XML". Web page, May 2000. http://www.megginson.com/SAX/index.html.

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

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

[22] João Pereira, Françoise Fabret, François Llirbat, Radu Preotiuc-Pietro, Kenneth A. Ross, and Dennis Shasha. Publish/subscribe on the Web at extreme speed. In Amr El Abbadi, Michael L. Brodie, Sharma Chakravarthy, Umeshwar Dayal, Nabil Kamel, Gunter Schlageter, and Kyu-Young Whang, editors, *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10–14, 2000, Cairo, Egypt*, pages 627–630, Los Altos, CA 94022, USA, 2000. Morgan Kaufmann Publishers.

[23] João Pereira, Françoise Fabret, François Llirbat, and Dennis Shasha. Efficient matching for web-based publish/subscribe systems. In *Conference on Cooperative Information Systems*, pages 162–173, 2000.

Appendix A

Sample IE schema XML file

```
<?xml version="1.0" encoding="UTF-8"?>
<IESCHEMA>
  <A>
    <VALUE>price</VALUE>
    <VALUE>period</VALUE>
    <VALUE>quantity</VALUE>
    <VALUE>furniture_category</VALUE>
    <VALUE>material</VALUE>
    <VALUE>year</VALUE>
    <VALUE>volume</VALUE>
    <VALUE>brand</VALUE>
    <VALUE>cc</VALUE>
    <VALUE>artist</VALUE>
    <VALUE>size</VALUE>
    <VALUE>event_type</VALUE>
  </A>
  <ET>
    <VALUE>antique</VALUE>
    <VALUE>furniture</VALUE>
    <VALUE>surf-board</VALUE>
    <VALUE>car</VALUE>
    <VALUE>painting</VALUE>
  </ET>
  <D>
    <DOMAIN oid="1" type="numeric">
      <OPER type="gt" />
      <OPER type="lt" />
      <OPER type="gteq" />
      <OPER type="lteq" />
      <OPER type="eq" />
```

```
    <OPER type="noteq" />
  </DOMAIN>
  <DOMAIN oid="2" type="string">
    <OPER type="contains" />
  </DOMAIN>
  <DOMAIN oid="3" type="enum" attr="bed-room ,living-room
                                          ,dining-room">
    <OPER type="eq" />
    <OPER type="noteq" />
  </DOMAIN>
  <DOMAIN oid="4" type="enum" attr="antique ,furniture ,
                               surf-board ,car ,painting">
    <OPER type="eq" />
    <OPER type="noteq" />
  </DOMAIN>
  <DOMAIN oid="5" type="enum" attr="wood ,carbon ,epox ,
                                  polyester ,paper ,glass">
    <OPER type="eq" />
    <OPER type="noteq" />
  </DOMAIN>
  <DOMAIN oid="6" type="enum" attr="opel ,volvo ,bmw ,
                                seat ,renault ,mazda ,mini">
    <OPER type="eq" />
    <OPER type="noteq" />
  </DOMAIN>
  <DOMAIN oid="7" type="string">
     <OPER type="contains" />
  </DOMAIN>
</D>
<DOM>
  <ATTR_DOM name="event_type" type="7" />
  <ATTR_DOM name="price" type="1" />
  <ATTR_DOM name="period" type="2" />
  <ATTR_DOM name="quantity" type="1" />
  <ATTR_DOM name="furniture_category" type="3" />
  <ATTR_DOM name="material" type="5" />
  <ATTR_DOM name="year" type="1" />
  <ATTR_DOM name="volume" type="1" />
  <ATTR_DOM name="brand" type="6" />
  <ATTR_DOM name="cc" type="1" />
  <ATTR_DOM name="artist" type="2" />
  <ATTR_DOM name="size" type="1" />
</DOM>
```

```
<E>
  <E_TYPE type="antique">
    <E_ATTR mandatory="true" unique="true" domid="1"
                                              type="price" />
    <E_ATTR mandatory="true" unique="true" domid="2"
                                             type="period" />
    <E_ATTR mandatory="true" unique="true" domid="1"
                                          type="quantity" />
    <E_ATTR mandatory="flase" unique="true" domid="5"
                                          type="material" />
    <E_ATTR mandatory="true" unique="true" domid="1"
                                            type="size" />
  </E_TYPE>
  <E_TYPE type="furniture">
    <E_ATTR mandatory="true" unique="true" domid="1"
                                              type="price" />
    <E_ATTR mandatory="true" unique="true" domid="1"
                                          type="quantity" />
    <E_ATTR mandatory="false" unique="true" domid="3"
                                type="furniture_category" />
    <E_ATTR mandatory="true" unique="true" domid="5"
                                          type="material" />
    <E_ATTR mandatory="true" unique="true" domid="1"
                                            type="size" />
  </E_TYPE>
  <E_TYPE type="surf-board">
    <E_ATTR mandatory="true" unique="true" domid="1"
                                              type="price" />
    <E_ATTR mandatory="true" unique="true" domid="1"
                                          type="quantity" />
    <E_ATTR mandatory="true" unique="true" domid="1"
                                            type="year" />
    <E_ATTR mandatory="false" unique="true" domid="5"
                                          type="material" />
    <E_ATTR mandatory="true" unique="true" domid="1"
                                            type="volume" />
  </E_TYPE>
  <E_TYPE type="car">
    <E_ATTR mandatory="true" unique="true" domid="1"
                                              type="price" />
    <E_ATTR mandatory="true" unique="true" domid="1"
                                          type="quantity" />
    <E_ATTR mandatory="true" unique="true" domid="1"
```

```
                                                    type="year" />
      <E_ATTR mandatory="false" unique="true" domid="6"
                                                    type="brand" />
      <E_ATTR mandatory="true" unique="true" domid="1"
                                                       type="cc" />
    </E_TYPE>
    <E_TYPE type="painting">
      <E_ATTR mandatory="true" unique="true" domid="1"
                                                    type="price" />
      <E_ATTR mandatory="true" unique="true" domid="1"
                                                type="quantity" />
      <E_ATTR mandatory="false" unique="true" domid="2"
                                                  type="period" />
      <E_ATTR mandatory="true" unique="true" domid="5"
                                                type="material" />
      <E_ATTR mandatory="true" unique="true" domid="1"
                                                    type="size" />
    </E_TYPE>
  </E>
</IESCHEMA>
```

Appendix B

Sample log XML file

```
<?xml version="1.0" encoding="UTF-8"?>
<USER_LOG>
  <USER username="sbile" password="sbileman">
    <SUBSCRIPTIONS>
      <PUBLISH_SUB id="1.0" username="sbile">
        <EVENT_TYPE>
          <VALUE type="car" oper="contains" />
        </EVENT_TYPE>
        <ATTRIBUTES>
          <ATTR type="price" oper="gteq">3</ATTR>
          <ATTR type="brand" oper="eq">bmw</ATTR>
        </ATTRIBUTES>
          <EVENTS>
            <PUBLISH_EVENT contact="sbile@hotmail.com">
              <EVENT_TYPE>
                <VALUE type="car" />
              </EVENT_TYPE>
              <ATTRIBUTES>
                <ATTR type="price">4</ATTR>
                <ATTR type="quantity">1</ATTR>
                <ATTR type="year">1</ATTR>
                <ATTR type="cc">22</ATTR>
                <ATTR type="brand">bmw</ATTR>
              </ATTRIBUTES>
            </PUBLISH_EVENT>
          </EVENTS>
      </PUBLISH_SUB>
    </SUBSCRIPTIONS>
  </USER>
  <USER username="automatic" password="automatic">
```

```
    <SUBSCRIPTIONS>
      <PUBLISH_SUB id="1.0" username="automatic">
        <EVENT_TYPE>
          <VALUE type="car" oper="contains" />
        </EVENT_TYPE>
        <ATTRIBUTES>
          <ATTR type="price" oper="gteq">4</ATTR>
          <ATTR type="quantity" oper="eq">4</ATTR>
          <ATTR type="year" oper="lt">1</ATTR>
          <ATTR type="brand" oper="noteq">mazda</ATTR>
          <ATTR type="cc" oper="gt">0</ATTR>
        </ATTRIBUTES>
        <EVENTS />
      </PUBLISH_SUB>
      <PUBLISH_SUB id="0.0" username="automatic">
        <EVENT_TYPE>
          <VALUE type="surf-board" oper="contains" />
        </EVENT_TYPE>
        <ATTRIBUTES>
          <ATTR type="price" oper="lt">6</ATTR>
          <ATTR type="quantity" oper="lt">6</ATTR>
          <ATTR type="year" oper="gteq">0</ATTR>
          <ATTR type="material" oper="noteq">epox</ATTR>
          <ATTR type="volume" oper="gt">3</ATTR>
        </ATTRIBUTES>
        <EVENTS>
          <PUBLISH_EVENT contact="sbile@intelligence.tuc.gr">
            <EVENT_TYPE>
              <VALUE type="surf-board" />
            </EVENT_TYPE>
            <ATTRIBUTES>
              <ATTR type="price">5</ATTR>
              <ATTR type="quantity">3</ATTR>
              <ATTR type="year">6</ATTR>
              <ATTR type="material">polyester</ATTR>
              <ATTR type="volume">6</ATTR>
            </ATTRIBUTES>
          </PUBLISH_EVENT>
        </EVENTS>
      </PUBLISH_SUB>
    </SUBSCRIPTIONS>
  </USER>
</USER_LOG>
```

a Sample user log file with two registered users

Appendix C

XML requests and DTDs

Add domain XML request & DTD

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ADD_DOMAIN SYSTEM
  "intellix.intelligence.tuc.gr:8090/dtds/add_dom.dtd">
<ADD_DOMAIN>
  <DOMAIN type="numeric">
    <OPER type="eq" />
    <OPER type="noteq" />
    </DOMAIN>
</ADD_DOMAIN>
```

```
<!ELEMENT ADD_DOMAIN (DOMAIN)>
<!ELEMENT DOMAIN (OPER+)>
  <!ATTLIST DOMAIN type ( numeric |
    string | enum ) #REQUIRED>
  <!ATTLIST DOMAIN attr CDATA "">
<!ELEMENT OPER EMPTY>
  <!ATTLIST OPER type ( eq | noteq
  | gt | lt | gteq | lteq | contains ) #REQUIRED>
```

Add attribute XML request & DTD

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ADD_ATTRIBUTE SYSTEM
  "intellix.intelligence.tuc.gr:8090/dtds/add_attr.dtd">
<ADD_ATTRIBUTE>
  <ATTR_DOM name="artist" type="1" />
</ADD_ATTRIBUTE>
```

```
<!ELEMENT ADD_ATTRIBUTE ( ATTR_DOM )>
<!ELEMENT ATTR_DOM EMPTY>
  <!ATTLIST ATTR_DOM name CDATA
    #REQUIRED>
  <!ATTLIST ATTR_DOM type
    (  1 | 2 | 3 | 4 | 5 | 6 | 7  ) #REQUIRED>
```

Extent event schema XML request & DTD

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE EXTENT_SCHEMA SYSTEM
  "intellix.intelligence.tuc.gr:8090/dtds/ext_scm.dtd">
<EXTENT_SCHEMA type="antique">
  <E_ATTR unique="true" type="size" />
</EXTENT_SCHEMA>
```

```
<!ELEMENT EXTENT_SCHEMA ( E_ATTR+ )>
  <!ATTLIST EXTENT_SCHEMA type ( antique
    | furniture | surf-board | car | painting ) #REQUIRED>
<!ELEMENT E_ATTR EMPTY>
  <!ATTLIST E_ATTR unique ( true | false ) #REQUIRED>
  <!ATTLIST E_ATTR type ( price | period | quantity
    | furniture_category | material | year | volume | brand | cc
      | artist | size | event_type ) #REQUIRED>
```

Add event XML & DTD

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ADD_EVENT SYSTEM
  "intellix.intelligence.tuc.gr:8090/dtds/add_event.dtd">
<ADD_EVENT>
  <E_TYPE type="paint">
    <E_ATTR mandatory="true" unique="true" domid="1"
      type="price" />
    <E_ATTR mandatory="true" unique="true" domid="2"
      type="period" />
    <E_ATTR mandatory="true" unique="true" domid="2"
      type="artist" />
    <E_ATTR mandatory="false" unique="true" domid="1"
      type="size" />
  </E_TYPE>
</ADD_EVENT>
```

```
<!ELEMENT ADD_EVENT ( E_TYPE )>
<!ELEMENT E_TYPE ( E_ATTR* )>
  <!ATTLIST E_TYPE type CDATA #REQUIRED>
<!ELEMENT E_ATTR EMPTY>
  <!ATTLIST E_ATTR mandatory ( true | false )
    #REQUIRED>
  <!ATTLIST E_ATTR unique ( true | false ) #REQUIRED>
  <!ATTLIST E_ATTR type ( price | period | quantity
    | furniture_category | material | year | volume | brand
      | cc | artist | size | event_type ) #REQUIRED>
```