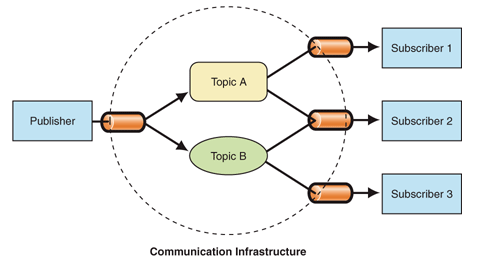
**Chapter 1**

**Introduction**

Publish/subscribe (pub/sub) is a paradigm for interconnecting information providers to information consumers in a distributed environment. Information providers publish information in the form of events to the pub/sub system, information consumers subscribe to a particular category of events within the system, and the system ensures the timely delivery of published events to all interested subscribers. A pub/sub system is typically implemented over a network of brokers that are responsible for routing events between publishers and subscribers.

****

[**Support**](javascript:void(0))

* 1. **Problem Definition**

To implement Fast Event matching algorithm for content based publish subscribe system using parallel approach and subscription covering relationship and analyze it under certain condition such as increase numbers of subscribers, publishers and throughput.

**1.2 Existing System**

Currently, the content-based publish/subscribe system is a hot research area. In this system the event is no longer dependent on an external standard (such as channel, theme, etc.) to classify, but the content in accordance with the event itself. Subscribers subscribe to the event according to the contents of the event, not to the subject of pre-defined themes by the system. When designing efficient content-based publish/subscribe systems, one of the key issues to be resolved is what strategy should be used to efficiently match large numbers of events and subscriptions.

Some industries uses a parallel search tree to solve the matching between the event and subscription. Each event agent has a complete parallel search tree, and determines the forwarding direction of the incident by means of the tree. But the algorithm can only deal with equal determination. Siena is a event notification service in the WAN, which uses the covering relation of the subscriptions and merging subscriptions to reduce the matching frequency, thus reducing the matching complexity.

* 1. **Project Objectives**
* Study the existing sequential content based publish/subscribe system.
* Parallelize that sequential approach.
* Implement Covering relationship for achieving fast matching.
* Analyze system under various parameter such as no of processors,no of inputs.

**Chapter 2**

**System Design**

* 1. **Subscription Language and Event Model**

For content-based publish/subscribe systems, on the one hand sufficient expression languages must be provided so that users can describe the interesting events conveniently, on the other hand the cost that it takes to match the event and subscription should be balanced. As a compromise, we use a subset of SQL language of relational database as the subscription language.

Event :

Event consists of a set of attributes {A1, A2, ... An}, and each attribute is a triple (data type, attribute name, value). Events can be expressed as: Event = ∪ attribute.

Constraints:

Constraint is a stateless Boolean expression. It’s expressed as quad (type, attribute, operator, value).

**Definition 1.**

If attribute A (typea, namea, valuea) of the event matches against a

Constraint C (typec, attributec, opc, valuec), the condition must be: (typea = typec)Λ (namea = attributec)Λ{ opc (valuea, valuec) = true}, denoted by A∈ *match* C.

**Definition 2.**

Given two constraints c1 and c2, we say that c2 covers c1, denoted by c1⊆c2, namely: (∀a: a∈ *match* c1→a∈ *match* c2)c1⊆c2.

Subscription :

Subscription is composed of a constraints set {C1, C2, ... Cn}. The relation between constraints is the conjunction relations, expressed as Subscription = ∪ constraint.

**Definition 3.**

For each constraint c in the subscription S, there is at least one attribute a in the event E, making a∈ *match* c, then we affirm that the event E and subscription S match, namely:∀c∈S, ∃ a∈E∧a∈ *match* cE∈ *match* S.

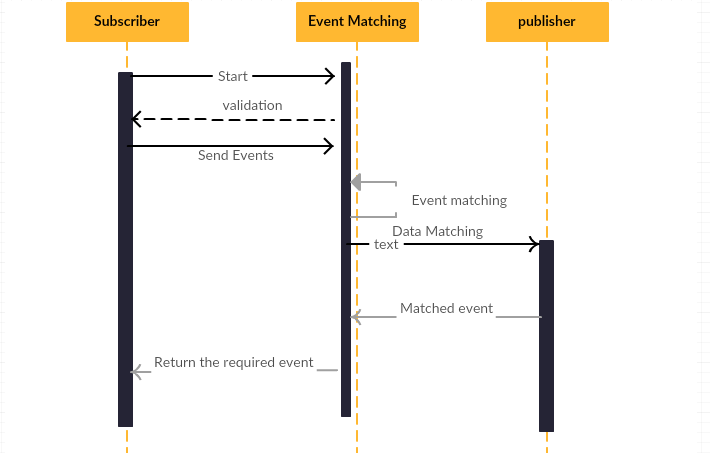
* 1. **UML Diagram**
     1. **Class Diagram**



* + 1. **Usecase Diagram**



**2.2.3**Sequence Diagram:



**Chapter 3:**

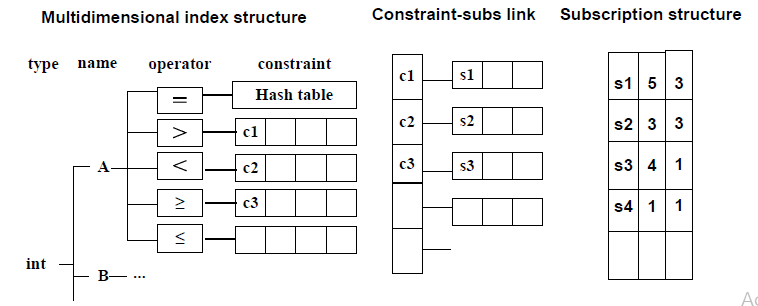
**Methodology**

**3.1 Matching Algorithm**

* **Data Structure**

The main data structure is shown in Figure 1. First, we break all the subscriptions into the constraint sets, remove duplicate constraints. Then these constraints are organized into a multidimensional indexing structure, in accordance with its data type, attribute name, comparison operators. Index is made level by level, and the final index entry comes into a list of constraints.

To maintain the subordination relation of the constraints and subscriptions, a constraint-subs link is taken, where each constraint points to a list that stores all the subscriptions identification that contains the constraint.



**Figure :** Data Structure of index count matching algorithm

**3.2 Algorithm Steps:**

The first stage is to find matching constraint for each attribute of the event according to their types and supported comparison operators; while the comparison operator is "=", do the hash table lookup directly, and for other comparison operators, get the first match of the constraints, thanks to the coverage relationship between constraints, its subsequent binding will also meet the match.

In the second phase, for each constraint has been successfully matched, the subscriptions belongs to it is searched. Then these subscriptions’ corresponding count value will plus 1. If the subscription’s count value is equal to the targe value, the subscription matches.

Algorithm:

Input : Dataset of Constraint and Event . Its near about 100000 records in constraint dataset.

1. Divide Constraints in a parts.
2. Read these Constraints and write in hashmap.

Algorithem:



**3.3 Praposed System:**

The parallel architecture is as shown in figure.

****

Parallelism is achieved through divide and conquer technique, which means that independent tasks are assigned to individual processors. Processing is done on each processor and respective result is generated. Final result will be the combination of results processed by each processor.

In our system, event matching will be done using parallelism ,hence many subscribers will get the message at the same time. As shown in fig. 2 in the first stage all the subscribers are shown. In the second stage, independent subscribers are decomposed into two processors and event matching is done at basic level. In the third stage, After event matching only interested subscribers will getting the messages at the same time. In figure S2 and s5 will get the message at same time.

**3.4 Subscription Covering Relationships:**

Covering relationships can be exploited to increase system efficiency as follows: If a newly arrived subscription s1 is covered by an existing subscription s2, then s1 is not forwarded as all notifications matching s1 are already being received. This reduces the number of subscriptions forwarded, without losing interesting notifications. Repeating this process at every intermediate router to which a subscription is forwarded can potentially reduce the number of forwarded subscriptions significantly.

The advantages are twofold:

* The number of subscription and unsubscription control messages is reduced
* The sizes of routing tables decrease.

While these advantages are well recognized and current solutions employ covering detection there has been little work on developing efficient algorithms for covering detection in a large database of subscriptions. We address this problem by proposing suitable indexing schemes.

**Indexing for Covering Detection**

We propose a way to index a large database of subscriptions to quickly detect covering relationships. The index is dynamic, i.e., it supports addition and deletion of subscriptions, and provides two operation .

* **Subscribe:**

When a new subscription s arrives, the indexing scheme is used to find if there is a current subscription covering s. If s is covered, it is not forwarded. Our index can find covering subscriptions (if any) by examining only a fraction of the subscriptions in the database.

* **Unsubscribe:**

When a previously issued subscription s is unsubscribed, it must be deleted from the database. As a result, some subscriptions which were previously covered by s may no longer be covered by any other subscriptions. These should be detected, and forwarded to other routers. Otherwise, the clients might miss interested notifications in the future. To facilitate this task, we need a separate data structure to maintain the currently detected covering relations among subscriptions.

We name this data structure the relation graph. Our solution is structured in two layers:

* The upper layer is the relation graph, which stores the currently detected covering relationships among subscriptions.
* The lower layer is the actual index, which is used to detect new covering relationships when subscriptions are added or deleted from the database.

1. **Relation Graph:**

There are many possible organizations for the relation graph, and we investigate the tradeoffs among these. We present a formal analysis to show that a simple structure for the graph, where one covering subscription is detected (if any) for each new subscription, provides a 2-approximation algorithm for optimizing the average total number of queries to the underlying index. To our knowledge, such an analysis has not been carried out so far. These theoretical findings are confirmed by experiments.

1. **Indexing Numeric Subscriptions:**

The design of the lower layer (the index) is specific to the format of subscriptions, and the expressiveness of the query language. we study an important special case where the different fields of the notifications are numeric values, and the subscriptions themselves are rectangular constraints on these numeric fields. Numbers are the most basic data type. In many applications, it is natural to describe data attributes by using numbers (examples are demographic census, stock prices). We believe that a good solution to the case of numeric attributes is significant in practice and also provides insight into the general problem.

**Chapter 4:**

**System Specification**

**4.1 Hardware**

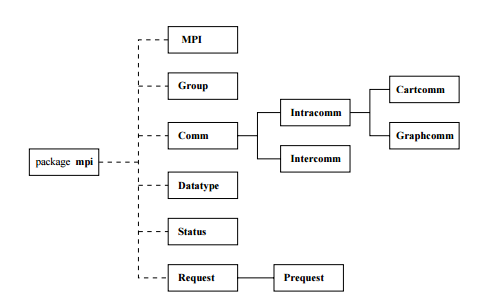
* RAM : Minimum 8GB
* Processor : Intel Pentium 4 and Above
  1. **Software** 
     1. **Language :**
* Java
* MPI Java

**MPI JAVA :**

MPI is not a programming language, but rather a standard library that is used to send messages between multiple processes. These processes can be located on the same system or on a collection of distributed servers. Unlike Open MP, the distributed nature of MPI allows it to work on almost any parallel environment. MPI is a speciation for the developers and users of message passing libraries. By It self, it is NOT a library - but rather the speciation of what such a library should be. MPI primarily addresses the message-passing parallel programming model: data is moved from the address space of one process to that of another process through co-operative operations on each process. Simply stated, the goal of the Message Passing Interface is to provide a widely used standard for writing message passing programs.

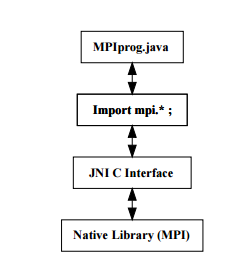
MPIJava is an object-oriented Java interface to the standard [Message Passing Interface](http://www.mcs.anl.gov/mpi/index.html) (MPI). The interface was developed as part of the HPJava project, but mpiJava itself does not assume any special extensions to the Java language - it should be portable to any platform that provides compatible Java-development and native MPI environments.

MPIJava is a software package that provides Java wrappers to a native MPI, through the Java Native Interface.



The class MPI only has static members. It acts as a module containing global services, such as initialization of MPI, and many global constants including the default communicator COMM WORLD. The most important class in the package is the communicator class Comm. All communication functions in mpiJava are members of Comm or its subclasses. As usual in MPI, a communicator stands for a 'collective object' logically shared by a group of processors. The processes communicate, typically by addressing messages to their peers through the common communicator. Another class that is important for the discussion below is the Datatype class. This describes the type of the elements in the message buffers passed to send, receive, and all other communication functions. Various basic datatypes are predefind in the package.

**MPI Software Layers:**



The interface attempts to be:

Practical

Portable

Efficient

Flexible

MPI Programming Model:



**Chapter 5**

**Implementation**

**5.1 Screenshot**

**Chapter 6**

**Conclusion**

In this way an efficient matching algorithm based on subscription language and event model is proposed, which uses multi-level indexing for high-speed matching of the event and related constraints and reduces the number of matching by the use of the cover relation of the constraints. Implemented algorithm and tested it with a variety of configurations, the experimental results show that the algorithm has better performance and strong scalability.

**Chapter 7**

**References:**

* + A Fast Matching Algorithm for Content-Based Publish/Subscribe Systems Tao Xue and Qi Jia(2004)
  + Aguilera, M., Strom, R., Sturman, D., Astley, M., Chandra, T.: Matching events in a content-based subscription system. In: PODC: 18th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (1999)
  + Carzaniga, A., Rosenblum, D.S., Wolf, A.L.: Design and Evaluation of a Wide-Area Event Notification Service. ACM Transactions on Computer Systems 19(3), 332–383(2001)
  + M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching events in a content-based subscription system. In *Proceedings of* the ACM Symposium on Principles of Distributed Computing (PODC), 1999.