Streaming matching of events

Dmitry Schitinin
Yandex.Classifieds
dima.schitinin@gmail.com

Vadim Tsesko
Yandex.Classifieds
mail@incubos.org

ABSTRACT

Streaming matching of events based on its content has many applications including customer subscriptions or recommendations for new ads. Customer subscriptions system requires minimal delay between event appearance and corresponding notifications. Such system should be horizontally scalable with regard to events and customers. There are many approaches to solving this task ranging from brute-force ones to utilizing some complex data structures. We present our approach integrated into several high-loaded production services at Yandex.Classifieds.

Keywords

Publish-subscribe; content-based matching; content-based filtering; parametric search

CCS Concepts

•**Information systems** → *Document filtering;* Service discovery and interfaces; •**Computer systems organization** → *Reliability; Availability; Redundancy;*

1. INTRODUCTION

Publish-subscribe paradigm has many advantages including loose coupling and horizontal scalability. According to [8] these systems can be classified by their key features. Among them are:

- Subject-based or content-based
- Event matching algorithm
- System architectrue

In subject-based pub/sub systems subscribers express their interest in terms of subjects or topics. In contrast content-based systems allow subscribers to register their requirement on receiving events in terms of event content.

Depending on pub/sub system type different event matching algorithms are used. For subject-based systems simple table lookup approaches are acceptable. Algorithms used for content-based matching are more sophisticated.

Customers subscriptions systems are naturally publish-subscribe systems with content-based events matching. In addition these systems require minimal delay between an event appearance and the corresponding notification. Such systems have to be horizontally scalable to handle more events and to service more customers.

There are many approaches to solving this task. Some of them focus on matching algorithm efficiency and other refer to system architecture for efficient distribution of events and delivery of notifications. As for our knowledge all these systems require holding all subscriptions in the main memory at each server of the system to perform matching and routing. The existing approaches are discussed in Section 5. It's obvious that the requirement to hold all subscriptions on each server limits system horizontal scalability and doesn't allow to run these systems on commodity hardware.

The main focus of this paper is on scalable distributed architecture that can be effectively deployed on commodity hardware.

We present an approach integrated into several high-loaded production services. Section 2 introduces some basic concepts and describes the events matching algorithm. We describe production subscriptions system design in Section 3. Some implementation details are discussed in Section 4. Related works are overviewed in Section 5. Possible improvements of the approach are described in Section 6. Section 7 concludes the paper.

2. MATCHING ALGORITHM

An event matching algorithm is the core of a content-based matching system.

Event is as a set of *terms* and a payload represented by opaque array of bytes. *Term* has a *name* and a *value*. Term name is a non-empty string. Term value may be of various kinds:

- **Fixed** value is represented by string
- **Range** value is a one-dimensional interval represented by pair of numbers, one of the borders may be absent
- **Vertex** value is a two-dimensional point on the plane represented by a pair of real numbers
- **Polygon** value is a two-dimensional polygon on the plane represented by a sequence of plane points

CEE-SECR '16, October 28 - 29, 2016, Moscow, Russian Federation
© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-4884-3/16/10…$15.00
DOI: http://dx.doi.org/10.1145/3022211.3022218

2.1 Coverage

There is a non-symmetrical relation among terms called *covers.* A term covers another one if their names are equal and their values cover according to the next natural rules:

- *Fixed* covers other *Fixed* if their underlying values are exactly equal
- *Range* covers *Fixed* if the Fixed value is numeric and is included into the Range interval
- *Range* covers other *Range* if their underlying intervals' intersection is non-empty
- *Vertex* covers other *Vertex* if their coordinates are exactly equal
- *Polygon* covers *Vertex* if the vertex is within the polygon

The model and the coverage relation can be extended further depending on the specific use case.

2.2 Filters

Events are passed through *Filter*s and may be accepted or rejected. *Filter* is defined as:

- **TermFilter (TF)** is determined by a *term.* It accepts an event if the event contains a term covered by the filter term.
- **AndFilter (AF)** contains a collection of filters and accepts events accepted by *all* of the underlying filters
- **OrFilter (OF)** contains a collection of filters and accepts events accepted by *some* of the underlying filters
- **NotFilter (NF)** contains a filter and accepts events rejected by the underlying filter

2.3 Filter by event

Consider an event and a set of filters. The task is to choose filters from the given set which accept the event. This problem is also known as "Content based matching" and there are some related works in [1, 9, 10, 6]. A naive solution is obvious: test in succession whether filter accepts the event or not. But with growing size of the set of filters this solution will stop working. A smarter solution with complexity depending on the number of different term names might be acceptable.

Having event E with terms Te_1, Te_2, ..., Te_n consider various filter sets. For brevity we will refer to TermFilter underlying term name and value as a filter name and a filter value.

Let all filters be *TermFilter*s containing terms Tf_1, Tf_2, ..., Tf_m. To select filters which accept event E we can take its first term Te_1 with name $Te_1.name$, find a filter from the set with the same name (containing term with the same name) and test whether its term value covers Te_1 value or not. By repeating this operation for the rest of the event terms Te_2, Te_3, ..., Te_n we will select all filters accepting event E. Note that we can find TermFilter with particular name in constant time using HashMap (HashDictionary) data structure.

Let us consider a single *AndFilter AF* containing several TermFilters TF_1, TF_2,..,TF_m. We need to test whether

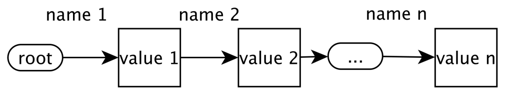

Figure 1: AndFilter as a path
The path represents `AF(TF(name 1, value 1), TF(name 2, value 2), ..., TF(name n, value n)`

event E is accepted by AF or not. Each TermFilter TF_i within AF should accept event E. We can introduce an order on terms' names (in simple case it may be the lexicographical one). For acceptance testing we take the first term using this order from event E. Denote it TE_{first}. Then try to take TermFilter TF_{first} from AF with the same name. The next cases are possible.

- TF_{first} exists and rejects term TE_{first}. It means that AndFilter AF rejects event E.
- TF_{first} exists and accepts term TE_{first}. It means that we should test acceptance of the term with the next name by the order.
- Filter doesn't exist. It means AndFilter AF doesn't impose a condition on the term and as above we should move on to the next name as in the previous case.

Having an order defined on terms names AndFilter AF may be represented as a path with edges marked with term names and vertices containing term values as shown on Figure 1.

Testing event E by filter AF can be represented as passing of the event through this path with gradual removal of terms from the event.

Having several AndFilters containing only TermFilters we can merge corresponding paths into a tree as shown on Figure 2. Term values sharing term name are grouped together. Each terminal path value is marked by the corresponding filter.

More formally, a tree *node* may be defined as a recursive structure with the next fields:

- `value` having term value type
- `filters` a set of filter IDs, possibly empty
- `children` a mapping from the term name into the set of *nodes*

2.4 Algorithm of descent

Having event E and merged AndFilters we can select the filters accepting E by the event "descending" down throught the tree starting from the root. Recall that there is an order on the term names. *Tree descent* recursive Algorithm 1 is applied to a tree *node* and a list of the event terms. Event terms appear in the list according to the defined order. *Filters* fields of all visited nodes contain IDs of filters which accept event E.

Consider an example. Let event E have the following terms:

- T_1 = `Term(name 1, value A)`
- T_2 = `Term(name 2, value D)`

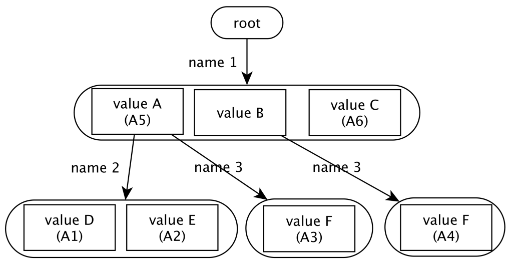

Figure 2: Merged AndFilters into a tree
This tree is the merge result of

- `A1 = AF(TF(name 1, value A), TF(name 2, value D))`
- `A2 = AF(TF(name 1, value A), TF(name 2, value E))`
- `A3 = AF(TF(name 1, value A), TF(name 3, value F))`
- `A4 = AF(TF(name 1, value B), TF(name 3, value F))`
- `A5 = AF(TF(name 1, value A))`
- `A6 = AF(TF(name 1, value C))`

Algorithm 1 Tree descent

```
 1: $result \leftarrow \emptyset$                    ▷ Set of accepting filters
 2: procedure DESCENT(node, terms)
 3:     $result \leftarrow result \cup node.filters$   ▷ Add all filters from the reached node to the accepting set
 4:     while terms is not empty do
 5:         $head \leftarrow terms.head$   ▷ Term with the next name by the order defined
 6:         $tail \leftarrow terms.tail$   ▷ Rest of the terms, may be empty
 7:         $DESCENT(node, tail)$   ▷ Descent without current term
 8:         $children \leftarrow node.children(head.name)$   ▷ Set of nodes with the same term name
 9:         for all child in children do
10:             if child.value covers head.value then
11:                 $DESCENT(child, tail)$
```

- T_3 = `Term(name 3, value F)`

Consider Figure 2. Using algorithm 1 event E descents through nodes with values A, D, F which means that filters $A1$, $A3$ and $A5$ accept the event.

An arbitrary filter can be transformed into some kind of *normal form*. Using disjunctive normal form with conjunctions *AndFilter*, disjunction *OrFilter* and *TermFilter* literals we can represent any filter in the form of

$$OF(AF_1(TF_1, .., TF_n), .., AF_m(TF_k, .., TF_l))$$

Thereby filters of any structure may be merged into the tree.

Term name ordering affects the form of the tree. It's feasible to choose filter term name order based on name frequency to make trees less branched to improve the descending algorithm performance.

2.5 Descent algorithm complexity

We now analyze the time required for event descending in Algorithm 1. Lines 9-11 select values from the group which covers given values. Suppose all the values are of Fixed type. As mentioned above in this case selection may be done in constant time. Under this restriction we measure event descent time by counting number of visited nodes. In case of k distinct term names maximum depth of the tree will be k. In the worst case on each tree level event will move through two branches (passing by current term and stripping it) so total number of visited nodes will be $2 + 2^2 + ... + 2^k$.

This exponential complexity is the upper bound of visited nodes count. Researches in [1] have proven that the very similar solution has an expected complexity that is sublinear in the number of subscriptions in the tree and have proposed some optimizations.

2.6 Two phase matching

In common case exponential complexity in the number of term names is not satisfactory for task solving. On the other hand, linear complexity of the naive solution is not acceptable too. Hovewer, exponential complexity is the worst case and as proven in [1] the algorithm has sub-linear complexity in the number of subscriptions.

The main idea of different approaches with matching trees is to use subscriptions commonalities to improve efficiency. The more commonalities subscriptions have the better an algorithm works. So it's not beneficial to add unique terms to the tree, for example, *PolygonTerms*. It's unlikely that two users will select the same polygon on the map while subscribing to new realty offers. However, the system must be able to handle such subscriptions.

We have designed a matching process in a following manner. The process is split into two phases: *candidate selection* and *final revision*. We fix a small number of the most frequently used term names and build the matching tree using only these terms. It's called *Candidates tree*. An event descending the tree matches subscription candidates that might accept the event. In revision phase we test explicitly whether resulted subscriptions match the event or not. This step has linear complexity in the candidate size.

2.7 Example

Here is a simple example of real world use case that can be easily expressed in terms of the described model and filtering algorithm.

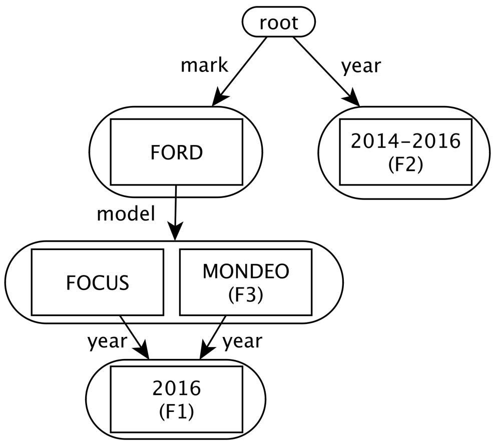

Figure 3: Tree of car ads filters

Consider a car selling web site. A car advertisement has a number of fields including mark, model, year of production and selling region. A customer can select offers by specifying its fields values using filters. These document fields and filters can be expressed using the following terms:

- `region` of type `Fixed`
- `mark` of type `Fixed`
- `model` of type `Fixed`

In addition a car ad can have Fixed term `year` whereas a corresponding term within filter may be of `Range` type since customer can specify a production year interval.

Consider three customer filters. Using HTTP query part notation they can be expressed as

- `F1: mark=FORD&model=FOCUS&model=MONDEO&year=2016`
- `F2: year_from=2014&year_to=2016`
- `F3: mark=FORD&model=MONDEO`

Translated into our notation these filters look like:

- `F1`:

```
AF(TF("mark", Fixed("FORD")),
    OF(TF("model",Fixed("FOCUS")),
        TF("model",Fixed("MONDEO"))),
    TF("year", Fixed("2016")))
```

- `F2`:

```
TF("year", Range(2014, 2016))
```

- `F3`:

```
AF(TF("mark", Fixed("FORD")),
    TF("model", Fixed("MONDEO)))
```

After `F1` normalization and merging with `F2` and `F3` we will get a tree on Figure 3.

Consider new car ads expressed as events. The first ad $A1$ consists of terms:

- `region: Point("Moscow")`
- `mark: Point("FORD")`
- `model: Point("MONDEO")`
- `year: Point("2016")`

The second car ad $A2$ contains the following terms:

- `region: Point("Saint-Petersburg")`
- `mark: Point("BMW")`
- `model: Point("3ER")`
- `year: Point("2015")`

$A1$ descending through the tree on Figure 3 using Algorithm 1 will visit nodes marked by filters $F1$, $F2$ and $F3$. It means that $A1$ satisfies to all of them.

Ad $A2$ visits only node marked by $F2$ hence satisfies only this filter.

3. SUBSCRIPTION SYSTEM

It's often convenient for customers to subscribe to interesting ads and receive notifications about them.

A naive implementation of this feature may include regular execution of saved search filters and analysis of results. Despite its simplicity the solution has many disadvantages such as extra (and often useless) load on search system and significant delay between a new offer appearance and the corresponding notifications.

On the other hand, a system providing this feature may be implemented on top of *streaming event matching* where events are new ads and filters are customer-specified conditions within subscriptions.

There are several functional requirements to the system:

- minimum delay between an ad appearance and the notification of the interested customers
- quick subscription creation and deletion
- ability to select preferred notification transport

Among the non-functional requirements we can highlight the following:

- horizontal scalability by the number of subscriptions
- horizontal scalability by the number of events to be matched
- fault tolerance

We have designed and implemented the system conforming to the specified requirements and described it below.

3.1 Architecture

Customers register *Subscription*s through the REST API provided by *API* component. A *Subscription* instance contains:

- `id` – unique identity
- `owner` – customer who created the subscription
- `filter` – filters specified by customers and expressed in concepts from Section 2

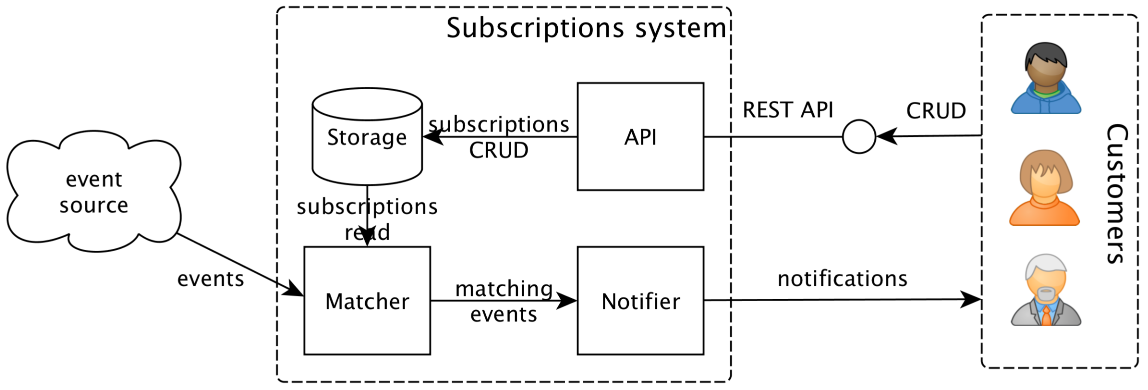

Figure 4: Architecture

- `delivery` – notifications delivery settings including preferred transport (e-mail, SMS, etc.) and scheduling ("send notifications every five minutes")

The web site has a component that identifies new ad appearance (*event source* on Figure 4). This component transforms new ads into events consisting of terms described in Section 2 and equipped with internal ad representation as a payload. The payload is useful for rendering notifications. Resulting events are passed to the *Matcher* component.

Matcher component is responsible for a subset of subscriptions. It receives produced *events*, finds corresponding subscriptions and emits tuples `<event, subscriptions>`. The tuple denotes that the `event` matches `subscriptions` and may be called *matching event*. *Matching events* are passed to the *Notifier* component.

Notifier is responsible for grouping matching event by subscription, composing notifications and sending them to the customer according to subscription delivery settings.

The data flow is shown on Figure 5.

3.2 Matcher

Matcher component is the core of the entire subscription system. It implements the content-based event matching mechanism. Receiving stream of events it produces stream of tuples `<event, subscriptions>`: providing a set of subscriptions interested in the `event`.

Under the hood Matcher contains filter-by-event mechanism described in Section 2. The filter tree is stored in the Matcher process main memory since matching event subscriptions should be very fast. As the number of subscriptions grows trees may become huge and might not fit into the main memory of a single machine. In this case we have to distribute the tree across several nodes. It's possible to do this since subscriptions are independent and having two or more filter trees requires only that event descend through each of them. Moreover it can be done in parallel.

Matcher component is a cluster of distributed peer processes containing a part of the entire set of subscriptions. Every matcher instance includes a broadcasting router that replicates received events to neighbours. Then all matchers process replicated event independently. This configuration is shown on Figure 6.

We have several data centers each containing event source. There is a Matcher cluster in each data center. Each cluster contains all subscriptions and instances within a cluster distribute subscriptions among each other. It's useful to place the Matcher cluster next to event source within data center to minimize inter-location traffic. However, in case of local Matcher cluster unavailability the event source will interact with the cluster in the nearest location. This intraction is performed through a system load balancing mechanism. In our case Haproxy[1] is used for this purpose.

3.2.1 *Subscription distribution*

Subscription distribution among matcher instances is implemented using *token* distribution. A *token* is a part of responsibility that is held by the *owner*. There is fixed amount of tokens in the system, typically tens or hundreds. *Token distributor* is described in Section 4.2.

Every matcher instance possesses some tokens, sees all token ownership map and knows the owner of each token. These ownership instances store some payload like API endpoints of the owner. Matcher routers use this date to perform service discovery of all matcher instances within the cluster.

Having got in possession token `t` a matcher becomes responsible for subscriptions with this token. It retrieves them from the storage and updates his internal tree of filters with the ones from the subscriptions retrieved.

In addition to Matcher component scalability the token distribution approach provides its reliability. Having crashed a matcher instance releases all the possessed tokens so that the peers can distribute the tokens among themselves thereby acquiring corresponding subscriptions and resuming event matching.

After matcher instance processes an event and finds the appropriate subscriptions it interacts with *Notifier* component by sending it matching events `<event, subscriptions>`.

3.3 Notifier

Notifier component groups received matching events `<event, subscriptions>` by subscription, composes notifications to the customers and sends them. Notifier uses the same approach to solve scalability and reliability issues as Matcher does. Notifier is also deployed cluster of distributed peer

[1]http://www.haproxy.org

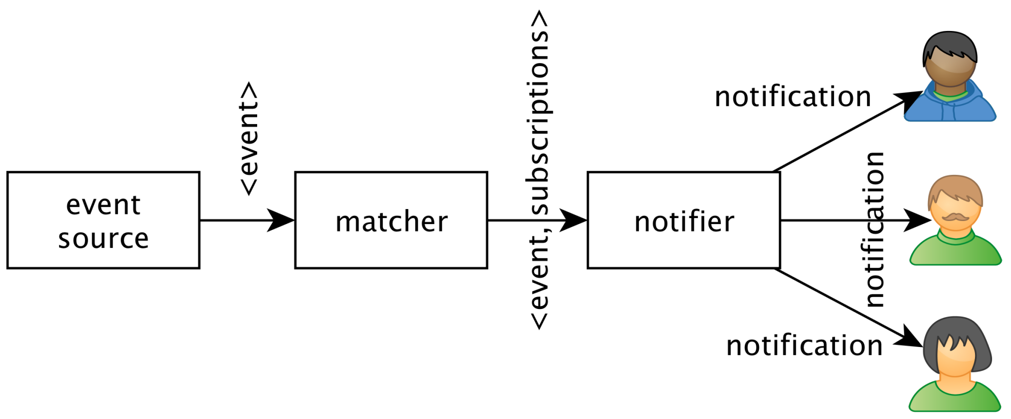

Figure 5: Subscriptions data flow

processes. Every notifier instance is responsible for a part of entire set of subscriptions assigned by tokens distribution algorithm. A notifier instance has a router that splits received matching event `<event, subscriptions>` into number of `<event, subscription>` events and routes them to the notifier instance responsible for the subscriptions (maybe himself). This configuration is shown on Figure 7.

After obtaining token `t` a notifier becomes responsible for aggregate events on the subscriptions with this token. Fault tolerance is implemented in the same way as in the Matcher cluster.

It is worth noting that tokens from the matcher environment and from the notifier one are two independent token set and there are distinct ownership maps and payloads.

3.4 API

API is a cluster of independent stateless processes providing HTTP REST API for subscriptions CRUD. Every CRUD operation relies on the persistent storage. In spite of regular storage polling by matcher instances API signals the appropriate matcher instance about specific subscription modifications to speed up the subscription emergence within Matcher.

3.5 Applications

We have integrated the implemented approach into our Web services including Auto.ru[2], Yandex.Realty[3], Yandex.Auto[4] and several internal subsystems.

Figure 8 shows matching events rate on our cluster and the corresponding latency for one of the services with 300K active subscriptions.

For performing stress testing we have use two data centers. Each data center contains event source and Matcher cluster consisting of two matcher instances. The obtained througput and latency is shown on Figure **??**. Figure **??** illustrates the throughtput and latency of a single matching instance form the cluster.

[2]https://auto.ru
[3]https://realty.yandex.ru
[4]https://auto.yandex.ru

4. REFINEMENTS

In this section we discuss some implementation details.

4.1 Filter tree implementation

We have implemented the model and filters from Section 2 as immutable data structures in Scala[5]. Immutability allows concurrent event descend and tree updates. We have adopted Scalacheck[6] for filter by event engine testing. This powerful tool helped us to find several tricky bugs in the implementation.

4.2 Tokens distributor

There are pieces of responsibility in the system that should be distributed among parties. A whole ”responsibility” can be expressed as a set of *tokens*. It's convenient to use integer numbers from interval $[0, n)$ to represent n tokens. Consider m parties among whom n tokens should be distributed in disjunctive and fair manner:

- there is no ownerless token
- there is no token with more than one owner
- difference between the richest owner and the poorest one is no more then one token

We have designed and implemented *Token distributor* approach to solve this task. There are three basic actions:

- `acquire(owner, token): bool` – tries to acquire ownerless *token* by *owner*; returns `true` if the acquisition has been successful and `false` otherwise (the *token* has been acquired by another owner)
- `release(owner, token): bool` – tries to release owned by *owner token*; returns `true` if *token* has been successfully released and `false` otherwise (the *token* is not possessed by *owner* anymore)
- `steal(owner, token, thief): bool` – *thief* tries to steal *token* from its current *owner*; returns `true` if

[5]http://www.scala-lang.org
[6]http://www.scalacheck.org

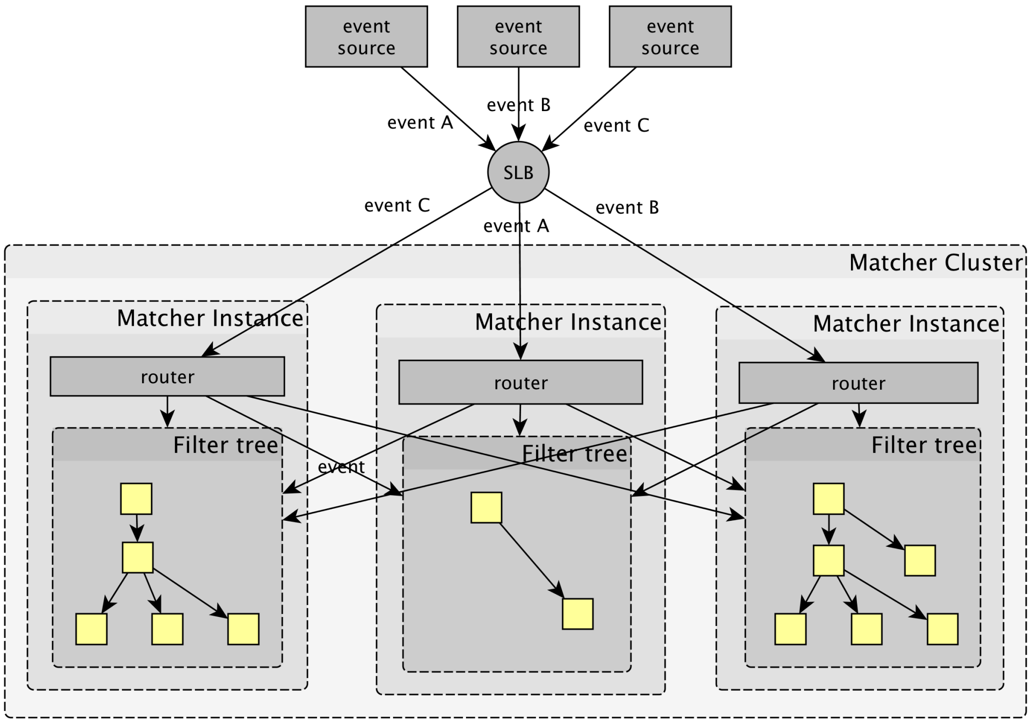

Figure 6: Matcher cluster

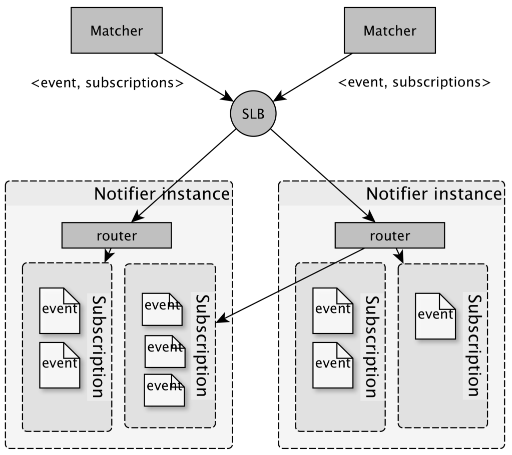

Figure 7: Notifier cluster

theft has been successfull (*token* was really possessed by *owner*) and `false` otherwise

Each party sees entire ownership map and can decide to do some actions depending on the map configuration. Party is considered to be:

- *rich* if there are no parties with greater number of owned tokens and there is a party with less number
- *poor* if there are no parties with less number of owned tokens and there is a party with greater number

To choose the next action a party uses the following rules:

- If the party is single and there is an ownerless token the party `acquire`s it
- It the party is poor and there is an ownerless token the party `acquire`s it
- If the party is poor and the distribution is not fair the party decides to `steal` a token from a one of the rich owners (choosen randomly)
- If all the parties have the same number of tokens and there is an ownerless token the party *acquire*s it

Distributed consensus system is required for implementation in distributed environment. We use [Apach Zookeeper][7] for this purpose.

4.3 Matcher and Notifier internals

When designing Matcher and Notifier we have adopted actor model implementation by Akka[8].

Having crashed a notifier instance may lose already accumulated events for subscriptions. To avoid this behavior notifier instances periodically dump their state to the persistent storage. Having got in possession token `t` a notifier instance restores the state of the corresponding subscriptions from the storage.

5. RELATED WORK

[7]http://zookeeper.apache.org
[8]http://akka.io

There are many results on content-based matching and related topics. They vary by matching algorithm efficiency and query language expressiveness.

The first work on matching algorithm based on matching trees is presented in [1]. The approach handles subscriptions containing conjunction attribute tests. The authors introduce Parallel Search Tree (PST) and calculate expected time and space complexity of the solution.

The related work [2] uses PST as a matching algorithm and focuses on efficient event multicasting to subscribers. To perform this matching and multicasting each broker contain the entire subscription set in the main memory.

An alternative matching algorithm is proposed in [7]. The authors describe an approach which uses indexing techniques for fast matching of atomic conditions and cluster subscriptions to minimize CPU cache misses while refining matched subscriptions. The approach demonstrates very high matching performance at the cost of the time needed to construct the matching data structue (tens of minutes or even hours for millions of subscriptions).

Researchers in [4] notice that approaches to matching in [1] and [7] are restricted to conjunctive subscriptions. This restriction leads to the exponential grow of the number of subscriptions to be submitted to matching mechanism in case of disjunctive condition presence in the initial subscriptions set. The authors generalize Parallel Search Tree to Binary Decision Diagram thus preventing exponential grow of the number of subscriptions without noticable performance penalty.

The technical report [5] describes Siena content-based notification service. The paper doesn't specify exact algorithm used for matching. The survey [8] says Binary Desicion Diagrams as in [4] are used for matching while the authors of [4] say that their filtering engine can be adapted to Siena.

Some ideas described in these papers might be used to improve matching performance of a subscription system. Hovewer all the approaches imply that subscriptions fit into the main memory of each node in contrast to our solution.

6. FUTURE WORK

The first candidate selection phase has predictable complexity in case of *Fixed* values. If values of more complex types occur within the tree more advanced techniques should be applied for selection of the next node (improvement of Algorithm 1 lines 9-11). Depending on the value type different indexing approaches may be used, including kd-trees [3] for ranges and polygons.

7. CONCLUSIONS

We described an approach to content-based event matching and its scalable distributed architecture. The solution is integrated into several production services. The described architecture allows for horizontal scalability by adding more nodes to handle more subscriptions and events. As a result of deploying the approach we got rid of the naive solution with regular requests to search system. Thereby we eliminated extra load on search system and increased quality of service for our customers: the latency between event appearance and the corresponding notifications is under several seconds.

8. REFERENCES

[1] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching events in a content-based subscription system. In *Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*. ACM, 1999.

[2] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. E. Strom, and D. C. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. In *Distributed Computing Systems, 1999. Proceedings. 19th IEEE International Conference on*, pages 262–272. IEEE, 1999.

[3] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[4] A. Campailla, S. Chaki, E. Clarke, S. Jha, and H. Veith. Efficient filtering in publish-subscribe systems using binary decision diagrams. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 443–452. IEEE Computer Society, 2001.

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

[6] F. Fabret, H.-A. Jacobsen, F. Llirbat, J. Pereira, and K. Ross. Filtering Algorithms and Implementation for Very Fast Publish/Subscribe Systems. In *SIGMOD Conference*, May 2001.

[7] F. Fabret, H. A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. In *ACM SIGMOD Record*, volume 30, pages 115–126. ACM, 2001.

[8] Y. Liu, B. Plale, et al. Survey of publish subscribe event systems. *Computer Science Dept, Indian University*, 16, 2003.

[9] M. Sadoghi and H.-A. Jacobsen. Location-based Matching in Publish/Subscribe Revisited. In *ACM/IFIP/USENIX 13th International Conference on Middleware*, 2012.

[10] M. Sadoghi and H.-A. Jacobsen. Adaptive Parallel Compressed Event Matching. In *30th IEEE International Conference on Data Engineering*, 2014.

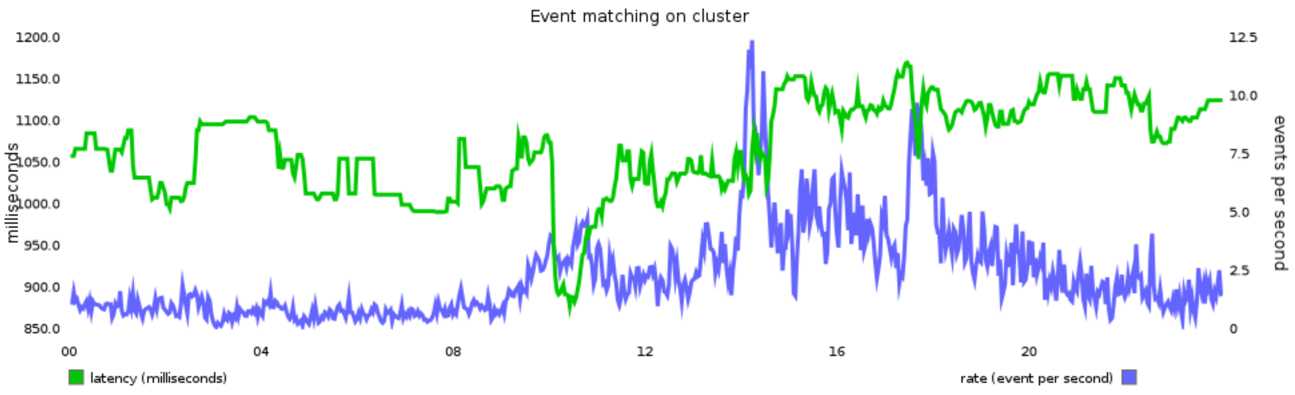

Figure 8: Events matching on a cluster

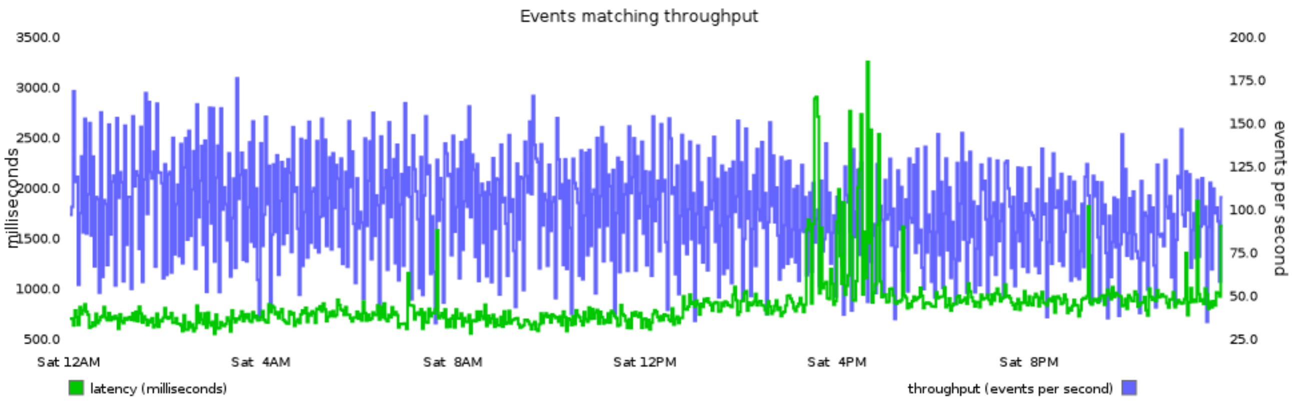

Figure 9: Events matching throughput on a cluster

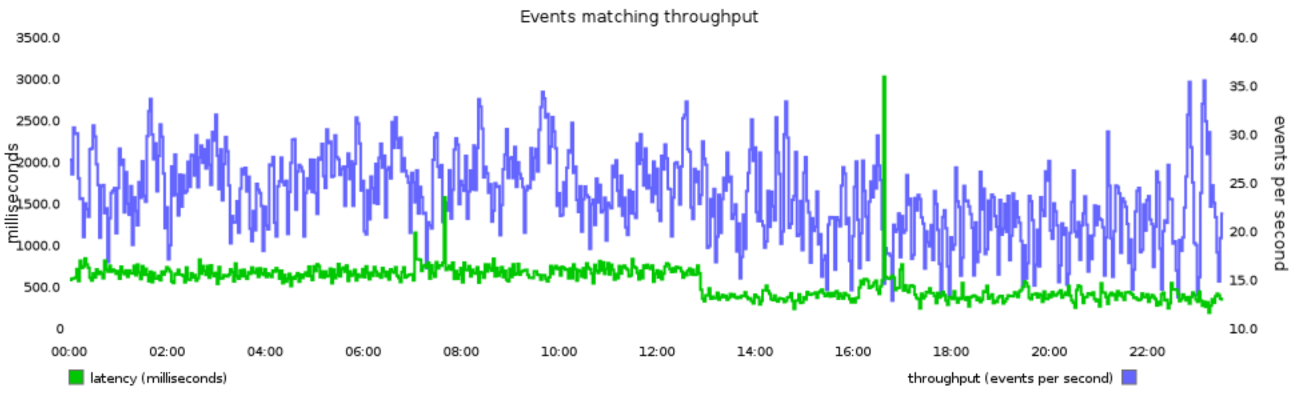

Figure 10: Events matching throughtput on a node