A Publish & Subscribe Architecture for Distributed Metadata Management*

Markus Keidl[1] Alexander Kreutz[1] Alfons Kemper[1] Donald Kossmann[2]

[1] Universität Passau
94030 Passau, Germany
⟨*last name*⟩@db.fmi.uni-passau.de

[2] Technische Universität München
81667 München, Germany
kossmann@in.tum.de

Abstract

The emergence of electronic marketplaces and other electronic services and applications on the Internet is creating a growing demand for effective management of resources. Due to the nature of the Internet such information changes rapidly. Furthermore, such information must be available for a large number of users and applications, and copies of pieces of information should be stored near the users that need this particular information. In this paper, we present the architecture of MDV, a distributed metadata management system. MDV has a 3-tier architecture and supports caching and replication in the middle-tier so that queries can be evaluated locally. Users and applications specify the information they need and that is replicated using a specialized subscription language. In order to keep replicas up-to-date and initiate the replication of new and relevant information, MDV implements a novel, scalable publish & subscribe algorithm. We describe this algorithm in detail, show how it can be implemented using a standard relational database system, and present the results of performance experiments conducted using our prototype implementation.

1 Introduction

Nowadays, the Web is one of the main driving forces behind the development of new and innovative applications. The emergence of electronic marketplaces and other electronic services and applications on the Internet is creating a growing demand for effective management of resources. Dynamic composition of such web services requires extensive metadata for the description, administration, and discovery of these services. Due to the nature of the Internet such information changes rapidly. Furthermore, such information must be available for a large number of users and applications, and copies of pieces of information should be stored near the users that need this particular information. Thus, metadata about such resources and services is a key to the success of these services.

In this paper, we present the architecture of the MDV system, a distributed metadata management system. MDV has a 3-tier architecture and supports caching and replication in the middle-tier so that queries can be evaluated locally, i.e., no expensive communication across the Internet is necessary. This supports fast discovery of metadata which is, e.g., necessary in web service composition or query optimization. Users and applications specify the information they need and that is replicated using a specialized subscription language. This reduces the amount of data that has to be queried locally resulting in a better query execution performance.

MDV implements a novel, scalable publish & subscribe algorithm to keep replicas up-to-date and initiate the replication of new and relevant information. Although the algorithm is described in the context of RDF [20] and MDV's subscription language it is also applicable to, e.g., XML [6] and the XQuery language [8], a direction which we are currently investigating. We describe this algorithm in detail, especially how it deals with the possibly huge set of subscription rules. We show how the algorithm can be implemented using a standard relational database system thereby taking advantage of their matured storing, indexing, and querying abilities.

The MDV system was developed as part of our ObjectGlobe project, an open and distributed query processing system for data processing services on the Internet. Therefore, we use ObjectGlobe as an example client of MDV. The goal of the ObjectGlobe project is to distribute query processing capabilities (including those found in traditional database systems) across

*This research is supported by the German National Research Foundation under contract DFG Ke 401/7-1.

the Internet. The idea is to create an open market place for three kinds of suppliers: *data providers* supply data, *function providers* offer query operators to process the data, and *cycle providers* are contracted to execute query operators. For a detailed description of the ObjectGlobe system see [5].

The remainder of this paper is structured as follows: Section 2 presents the MDV system, its architecture, and core components. Section 3 describes our publish & subscribe algorithm, particularly the filter algorithm. Performance experiments conducted using our prototype implementation are presented in Section 4. Section 5 discusses some related work and Section 6 concludes this paper.

2 Overview of the MDV System

In this section, we describe the architecture of the MDV metadata management system and give a general overview of the system's components. Main features of our system are a 3-tier architecture that eases the adjustment to varying workloads and activity hot spots, efficient metadata access using caching, and a publish & subscribe mechanism for preserving cache consistency. MDV uses RDF [20] as its data model (using the XML syntax for documents) and RDF Schema [7] to define the schema the RDF metadata must conform to.

2.1 An example

Figure 1 shows an excerpt from an RDF document `doc.rdf`. The excerpt defines two resources: a `CycleProvider` and a `ServerInformation`. The former is a server on the Internet capable of executing arbitrary ObjectGlobe services, the latter contains information about the computer running the provider. The `rdf:ID` property defines a local identifier for each resource, `host` and `info` in the example. A unique identifier, called *URI reference*, is constructed by combining the local identifier of a resource with the (globally unique) URI associated with an RDF document. The CycleProvider resource contains three further properties: `serverHost` which contains the server's DNS name, `serverPort` which contains the provider's port, and `serverInformation` which is a reference to the ServerInformation resource storing data about the computer running the provider. It contains the size of the computer's main memory in MB (property `memory`) and the speed of its CPU in MHz (property `cpu`). Properties (like serverInformation) always reference resources using their URI reference, i.e., RDF does not distinguish between nested and referenced resources. So it is irrelevant if resources are defined as nested elements (as in Figure 1) or somewhere else in the same or even in another document.

```
<CycleProvider rdf:ID="host">
  <serverHost>pirates.uni-passau.de</serverHost>
  <serverPort>5874</serverPort>
  <serverInformation>
    <ServerInformation rdf:ID="info" memory="92"
                       cpu="600" />
  </serverInformation>
</CycleProvider>
```

Figure 1. Excerpt: MDV RDF document

2.2 Architecture Overview

Figure 2 depicts the MDV system's 3-tier architecture, consisting of *Metadata Providers*, *Local Metadata Repositories*, and *MDV clients*.

Metadata Providers (MDPs), referred to as *(MDV) backbone*, are distributed all over the Internet to provide a uniform access regarding network latency *and* metadata content. MDPs accomplish the latter by sharing the same schema and consistently replicating metadata among each other. Basically, the backbone is an extension of a distributed DBMS with a flat hierarchy, full synchronization, and replication.[1] All metadata stored at MDPs is regarded as global and publicly available.

Local Metadata Repositories (LMRs) are the components of the MDV system that do the actual metadata query processing. For efficiency reasons, i.e., to avoid communication across the Internet, LMRs cache global metadata and use only locally available metadata for query processing. Consequently, LMRs should be running close to the applications querying the metadata, e.g., in the same LAN. The cache of an LMR should contain relevant metadata, appropriate to the users or applications using it. LMRs use a publish & subscribe mechanism to fetch relevant metadata from an MDP and to receive updates to their data, that is, to keep their caches consistent. When *subscribing* to an MDP an LMR registers a set of subscription rules which define the parts of the MDP's metadata the LMR wants to cache. MDPs use subscription rules to *publish* updates, insertions, or deletions in the metadata to LMRs. In addition to global metadata, LMRs store local metadata that should not be accessible to the public and therefore is *not* forwarded to the backbone. Local metadata must be explicitly marked as such at registration time.

Applications and users accessing the MDV system are referred to as *MDV clients*. MDV clients can query metadata at an LMR using MDV's (declarative) query language which is not presented because of paper length limitations. It is quite similar to the rule language that

[1] A more sophisticated distributed architecture regarding partitioning and replication is possible in the backbone. But this is not the focus of our work. Work on partitioning and replication for distributed database systems is described in [24].

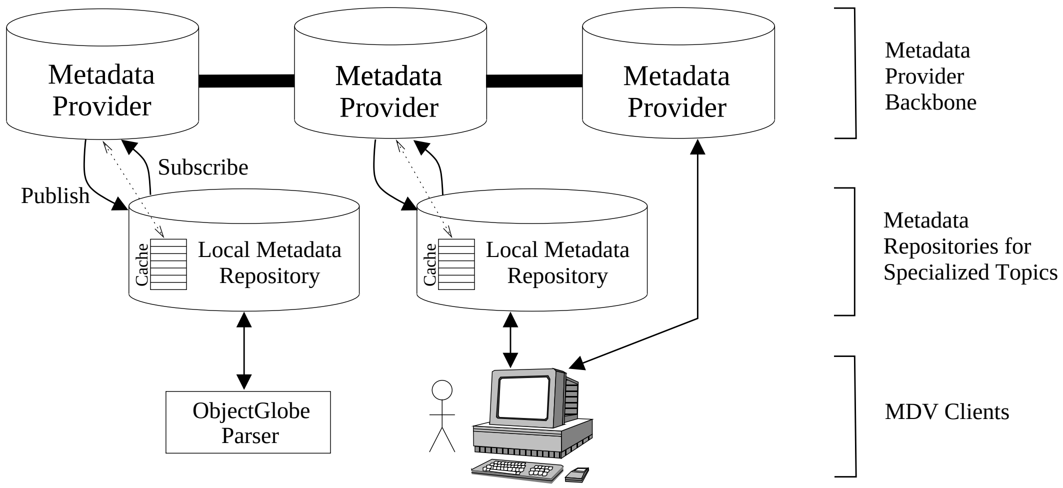


Figure 2. Overview of MDV's Architecture

is explained in Section 2.3. Real users can also browse metadata at an MDP (as depicted in Figure 2) and select it for caching. Their LMR will generate appropriate rules and update its set of subscription rules.

Metadata administration, i.e., registering new metadata and updating or deleting old metadata, is done at MDPs. New metadata must be registered within a valid RDF document; updating metadata essentially means re-registering a modified version of an already registered document. Deleting can be done either by removing parts from a document and updating it or by removing the complete document with all its content. This is the only way to add, update, or delete metadata. MDV's query language does not provide any update or delete functionality.

MDV is implemented in Java, so that it is portable which allows installation with very little effort, and it uses a relational database management system (RDBMS) as basic data storage. RDF documents are mapped to tables as described in [14]. Search requests are translated into SQL join queries. This translation is not one-to-one as MDV hides the details of how the metadata is stored. Translating search requests into SQL queries is quite complicated (albeit straightforward) and describing all the details is beyond the scope of this paper.

2.3 The Rule System

Subscription rules are specified by users browsing and selecting metadata or by administrators of LMRs. Rules must describe the kind of metadata that local MDV clients are interested in because only metadata matching the rules is cached locally and used for metadata query evaluation. Currently MDV uses a proprietary rule language which supports path expressions and joins. A rule is (informally) defined using the following SQL-like syntax:

`search` *Extension* e `register` e `where` *Predicates*(e)

This rule matches or registers all resources e that are an element of extension *Extension*—which is either some class defined in the schema or another subscription rule— and that satisfy the rule's `where` part. *Predicates* is a conjunction of elementary predicates where each is of the form $X \circ Y$ with X and Y either constants or valid path expressions (according to the schema) and $\circ \in \{=, !=, <, <=, >, >=,$ `contains`$\}$. MDV provides a special *any* operator `?` that can be applied to set-valued properties.[2] Currently our implementation does not support an `or` operator, but rules containing it can be split up easily into rules without it using the algebra of logic and negated operators.

Example 1 The following rule subscribes to all resources that are an instance of class CycleProvider, which must be defined in the schema, and that have a property serverHost that contains 'uni-passsau.de' and a serverInformation property that references a ServerInformation resource with a memory property value greater than 64:

`search`	CycleProvider c
`register`	c
`where`	c.serverHost `contains` 'uni-passau.de'
	`and` c.serverInformation.memory $>$ 64

So, this rule subscribes to all cycle providers that run on computers in the 'uni-passau.de' domain with more

[2]With set-valued properties all set elements must have the same type, even though RDF does not formally enforce this.

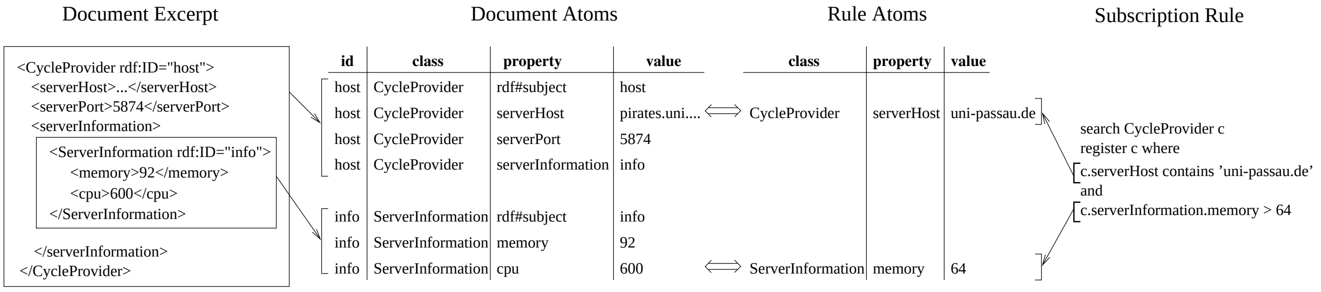


Figure 3. Basic Idea of the Filter Algorithm

than 64MB of main memory. For example, the CycleProvider resource defined in the document excerpt of Figure 1 matches this rule.

2.4 References

Example 1 shows one problem: The rule, applied to the metadata of Figure 1, will register the CycleProvider resource with reference `host` and transmit it to an LMR. But obviously the referenced ServerInformation resource must be transmitted, too. Otherwise, the CycleProvider resource will contain a dangling reference. So, what to do with referenced resources? Possible solutions are a) to never transmit them with a resource, b) to follow all references until no new references are found (i.e., calculating the closure), or c) to do something in between. The first two solutions both have major drawbacks, ranging from dangling references to a possible transmission of the complete database. Therefore, we chose to introduce *strong* and *weak* references. Resources referenced by the former are always transmitted together with the referencing resource whereas resources referenced by the latter type are never transmitted. MDV augments RDF schema with the necessary RDF properties to allow the definition of strong and weak references. The decision which references are strong and which are weak is part of the schema design and therefore the according designer carries the responsibility for defining them.

With strong references an LMR can receive resources where there is no corresponding rule for. An LMR must take care for deleting such resources if the resource that caused their transmission is deleted, e.g., because the according rule is changed or removed. MDV uses a *garbage collector* (based on reference counting) to detect such resources and to remove them if necessary.

3 The Publish & Subscribe Algorithm

In this section, we describe our publish & subscribe algorithm, particularly one of its core components, the filter algorithm. One of the main challenges in publishing of data is the evaluation of subscription rules. The evaluation is necessary to obtain all subscribers that have to be notified of new, updated, or deleted data. To avoid the evaluation of the possibly huge set of *all* subscription rules our filter determines a (small) subset of them that are at most affected by the modification of the data. Additionally, our filter takes advantage of rule/predicate redundancy and evaluates affected rules incrementally, i.e., using only the modified data as far as possible.

Our filter algorithm is solely based on standard relational database technology, mainly for the advantages in robustness, scalability, and query abilities. We implemented a prototype based on the MDV system, its rule language, and the RDF data model.

3.1 Overview of the Approach

Consider the following rule that registers all cycle providers (i.e., their resources) that are running in the domain 'uni-passau.de':

search CycleProvider c **register** c
where c.serverHost **contains** 'uni-passau.de'

This rule must be evaluated when a resource of class CycleProvider is registered, updated, or deleted. The following rule that registers all cycle providers that are running on a computer with more than 64MB of memory shows that it is not that simple:

search CycleProvider c **register** c
where c.serverInformation.memory > 64

This rule must be evaluated not only when a *CycleProvider* resource is registered, updated, or deleted, but also when the referenced *ServerInformation* resource is updated. For example, if the ServerInformation resource's memory property is updated from 32 to 128, all CycleProvider resources referencing the updated resource are afterwards matching the above rule.

Figure 3 illustrates the basic idea of our filter algorithm. Both, documents and subscription rules are decomposed into so-called *atoms*, i.e., basically tuples of a table. For the former, an atom is basically an RDF

FilterData

uri_reference	class	property	value
doc.rdf#host	CycleProvider	rdf#subject	doc.rdf#host
doc.rdf#host	CycleProvider	serverHost	pirates.uni-passau.de
doc.rdf#host	CycleProvider	serverPort	5874
doc.rdf#host	CycleProvider	serverInformation	doc.rdf#info
doc.rdf#info	ServerInformation	rdf#subject	doc.rdf#info
doc.rdf#info	ServerInformation	memory	92
doc.rdf#info	ServerInformation	cpu	600

Figure 4. Table *FilterData*, based on the RDF document of Figure 1

statement (or triple, as described in [20]). For the latter, an atom is composed of the rule parts that refer to a single class. The filter algorithm joins the document atoms with the rule atoms obtained from the subscription rule base to determine all rules that may register resources of the document and, therefore, have to be evaluated.

In summary, our publish & subscribe algorithm proceeds in three steps: 1) Newly registered documents must be decomposed. 2) Newly registered rules must be decomposed. 3) Document atoms and rule atoms are matched and rules that may register new resources are evaluated incrementally. We describe each of these steps in the following subsections.

3.2 Decomposition of Documents

Any newly registered RDF document is decomposed into its atoms, i.e., RDF statements as described in [20]. These statements and some additional information (necessary for filter execution) are inserted into the table *FilterData* (see Figure 4 for an example). Additionally, for each resource a tuple is inserted containing the URI reference and the class name (with property set to `rdf#subject` and value set to the resource's URI reference). Thus, rules are able to register a single resource using its URI reference.

3.3 Decomposition of Rules

To obtain the rule atoms a new subscription rule is processed in three steps: First, the rule is normalized, basically to ease decomposition. Afterwards, the normalized rule is decomposed into so-called *atomic rules*. Finally, a dependency tree is created based on information obtained from the decomposition step and merged with a global dependency graph. We distinguish two types of atomic rules: A *triggering rule* refers to a single class; it requires no results of other atomic rules and contains no path expressions, i.e., it contains only accesses to properties. A *join rule* represents a join of two extensions with a join predicate. It contains no other predicates and it always depends on two other atomic rules.

The normalization of rules is not crucial for the correctness of rule decomposition, but it eases its explanation and implementation. We call a rule normalized if its `search` part contains all classes that are used in its `where` part, not only directly using a variable but also in path expressions. Path expressions are not allowed in normalized rules, only accesses to properties (including the `?` operator), and they are split up therefore. As an example, we present the normalized form of the rule from Example 1:

```
search     CycleProvider c, ServerInformation s
register   c
where      c.serverHost contains 'uni-passau.de'
           and c.serverInformation = s
           and s.memory > 64
```

3.3.1 Rule Decomposition

The decomposition of a subscription rule into atomic rules is done based on the predicates used in it: In a first step, all predicates with a constant are removed from the original rule and for each of them a triggering rule is created with the predicate as `where` part, i.e., the triggering rule matches all resources that satisfy the predicate. If there are classes in the `search` clause without such a predicate, a triggering rule without `where` clause is created. After this, the original rule is adjusted to use the results of the triggering rules as input instead of evaluating the removed predicates. As an example, consider the (normalized) rule

```
search     CycleProvider c, ServerInformation s
register   c
where      c.serverHost contains 'uni-passau.de'
           and c.serverInformation = s
           and s.memory > 64 and s.cpu > 500
```

All predicates with constants are considered and appropriate triggering rules are generated:

```
search ServerInformation s register s
where s.memory > 64                              (RuleA)

search ServerInformation s register s
where s.cpu > 500                                (RuleB)

search CycleProvider c register c
where c.serverHost contains 'uni-passau.de'      (RuleC)
```

The original rule is adjusted afterwards:

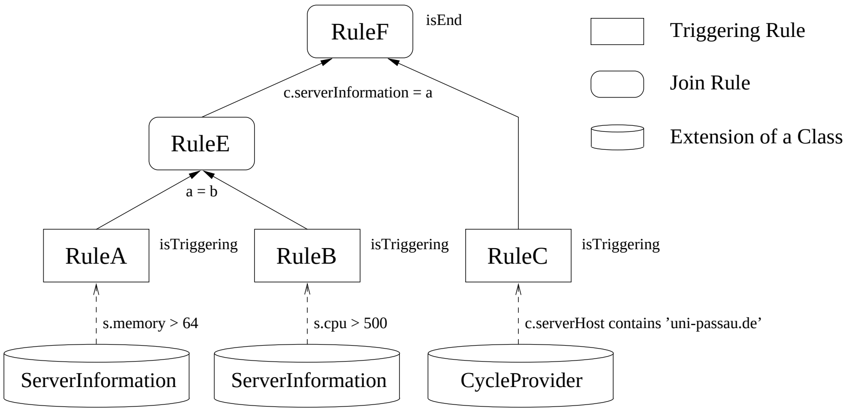


Figure 5. Dependency Tree of Example Rule in Section 3.3.1

`search` RuleA a, RuleB b, RuleC c `register` c
`where` a = b `and` c.serverInformation = a (RuleD)

Note, that a rule's *type* is the type of the resources it registers, e.g., the type of RuleD is CycleProvider. All remaining predicates in the original rule are join predicates. Now, subsequently each such predicate is removed and a join rule is created with the removed predicate in its `where` part. The original rule is again adjusted. This is done until the original rule is itself a join rule. In our example, two join rules are generated:

`search` RuleA a, RuleB b `register` a
`where` a = b (RuleE)

`search` RuleE a, RuleC c `register` c
`where` c.serverInformation = a (RuleF)

The subscription rule is now decomposed into the atomic rules RuleA, RuleB, RuleC, RuleE, and RuleF.

3.3.2 Creation of the Dependency Graph

The decomposition always creates non-cyclic dependencies between the generated atomic rules. These dependencies are represented in a dependency tree in which nodes represent atomic rules and directed edges represent dependencies. The tree contains an end rule (an atomic rule that produces the result of the subscription rule) as root node, one or more triggering rules as leave nodes, and join rules as inner nodes. Figure 5 depicts a dependency tree that is based on the atomic rules in Section 3.3.1.

After decomposition the generated atomic rules are merged with already existing atomic rules (stemming from previously registered rules). This is equivalent to merging the dependency tree of the newly registered rule with the *global dependency graph* which is a directed, acyclic graph that consists of the merged dependency trees of previously registered rules. By merging dependency trees the filter algorithm takes advantage of rule and predicate redundancy and, as a consequence, evaluates equivalent rules and atomic rules only once.

3.3.3 Rule Groups

Although the decomposition algorithm already considers redundancies, there remain similar atomic rules. Consider the following example:

`search` CycleProvider c `register` c
`where` c.serverInformation.memory $>$ 64

`search` CycleProvider c `register` c
`where` c.serverInformation.cpu $>$ 500

Decomposition results in the following atomic rules (note, that RuleA is already shared):

`search` CycleProvider c `register` c (RuleA)

`search` ServerInformation s `register` s
`where` s.memory $>$ 64 (RuleB1)

`search` RuleA c, RuleB1 s `register` c
`where` c.serverInformation = s (RuleC1)

`search` ServerInformation s `register` s
`where` s.cpu $>$ 500 (RuleB2)

`search` RuleA c, RuleB2 s `register` c
`where` c.serverInformation = s (RuleC2)

Comparing RuleC1 and RuleC2 reveals that both atomic rules have equal `register` and `where` parts and that even the classes the variables are bound to are equal.[3] But the resources used to evaluate the rules are different. To avoid individual evaluation of such join rules, *rule groups* are introduced. A rule group combines join rules which have an equal `where` part and where the corresponding variables are bound to the same class. All grouped join rules are evaluated at once by combining their input data, evaluating the shared `where` part, and splitting up the result afterwards. Figure 6 depicts this for the above example.

[3] Note that variable names need not be equal.

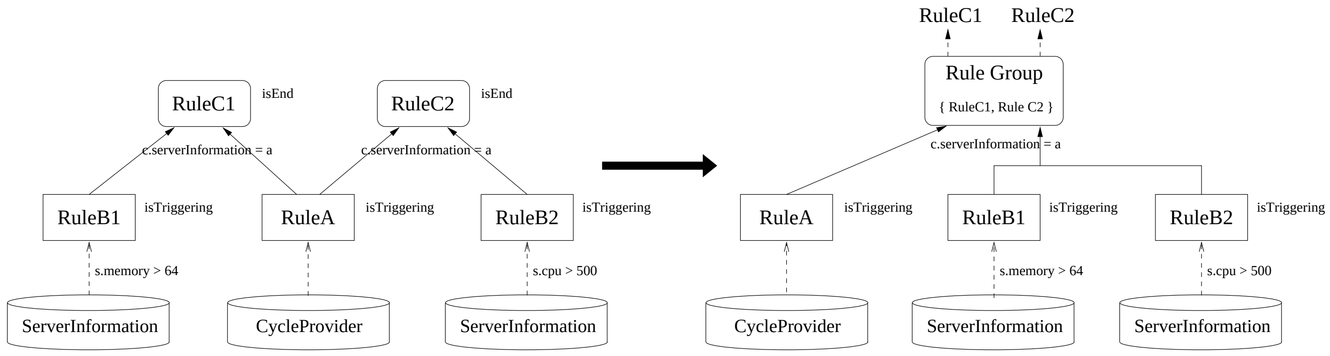


Figure 6. Generation of Rule Groups

3.3.4 Implementation Issues

We now describe the most important tables used by the filter algorithm. For brevity reasons, we omit tables that are not directly related to it, e.g., tables that store which rules an LMR registered. A key concept to an efficient filter implementation is the physical database design. First, the filter tables are used as indexes to all triggering rules affected by newly registered metadata. Given some metadata the tables allow an efficient determination of all triggering rules that have a `where` part that evaluates to true given the new metadata. Second, the tables themselves are created with indexes supporting an efficient access on the database level.

Table *AtomicRules*[4] stores all atomic rules (see Figure 7 for an example). There are no duplicates, i.e., no rules having the same rule text but different rule_ids. *RuleDependencies* stores the global dependency graph. A reference to the rule group of an atomic rule is stored in its *AtomicRules* tuple and, for efficiency reasons, in *RuleDependencies*, in tuples where the atomic rule is the target. The rule groups themselves are stored in *RuleGroups*.

Triggering rules are additionally inserted into one of the tables *FilterRules*<u>*OP*</u> or *FilterRules* depending on the operator used in their `where` part. Our current implementation supports comparisons with operators `<`, `<=`, `>`, and `>=` only on numerical constants. To avoid the creation of an own *FilterRules*<u>*OP*</u> table with an appropriate type of the value attribute for all numerical types, constants are stored as strings and reconverted when joining the tables with the *FilterData* table. Figure 8 shows an example extension of the *FilterRules/FilterRules*<u>*OP*</u> tables, based on the triggering rules from Section 3.3.1.

[4]We use view(rule_id, class) to refer to another atomic rule (instead of, e.g., RuleA).

3.4 The Filter Algorithm: Matching Documents and Rules

The filter algorithm is started after a new document was registered and decomposed. It consists of two steps: First, all triggering rules are determined that are affected by the registration of new metadata. Subsequently, all join rules depending on the affected triggering rules are evaluated incrementally, as defined by the global dependency graph.

Determination of Affected Triggering Rules It is crucial that a triggering rule refers to a single class. Its `where` part is either empty or a comparison with a constant. The former type of atomic rule matches any newly registered resource which is an instance of the class the rule refers to. For the latter type, the rule's predicate has to be evaluated based on the atoms of the document. One matching atom is sufficient for a triggering rule to be affected. Our prototype implementation starts with joining the table *FilterData* with *FilterRules* and all *FilterRules*<u>*OP*</u> tables using a join predicate depending on the actual *FilterRules/FilterRules*<u>*OP*</u> table. The left table of Figure 9 shows the result of this step based on the previous examples. (The table *ResultObjects* always contains the result of a filter step.)

Evaluation of Join Rules Now, all join rules depending on affected triggering rules are evaluated. With join rules complete incremental evaluation is not possible, so the results of atomic rules join rules depend on are materialized. The evaluation consists of several iterations. In each iteration all atomic rules depending on the atomic rules currently stored in *ResultObjects* are determined using the table *RuleDependencies*. Then, the rule groups of these atomic rules are evaluated using the resources currently stored in *ResultObjects* and—if necessary—materialized data as input. The result of this evaluation, i.e., the matching resources and the

AtomicRules

rule_id	text	group
1	`search` ServerInformation s `register` s `where` s.memory $>$ 64	
2	`search` ServerInformation s `register` s `where` s.cpu $>$ 500	
3	`search` CycleProvider c `register` c `where` c.serverHost `contains` 'uni-passau.de'	
4	`search` view(1, ServerInformation) a, view(2, ServerInformation) b `register` a `where` a = b	1
5	`search` view(4, ServerInformation) a, view(3, CycleProvider) c `register` c `where` c.serverInformation = a	2

RuleDependencies

source	target	param	group
1	4	1	1
2	4	2	1
4	5	1	2
3	5	2	2

RuleGroups

group	text
1	`search` group(ServerInformation) a, group(ServerInformation) b `register` a `where` a = b
2	`search` group(ServerInformation) a, group(CycleProvider) c `register` c `where` c.serverInformation = a

Figure 7. *AtomicRules*, *RuleDependencies*, **and** *RuleGroups*, **based on Section 3.3.1**

FilterRulesGT

rule_id	class	property	value
1	ServerInformation	memory	64
2	ServerInformation	cpu	500

FilterRulesCON

rule_id	class	property	value
3	CycleProvider	serverHost	uni-passau.de

Figure 8. Triggering Rules of Example 3.3.1

atomic rules they match, are again stored in *ResultObjects* and used as input of the next iteration. Any resources matching an end rule are saved for later. The algorithm terminates if there are no more dependent join rules. Termination is guaranteed because the dependency graph is an acyclic, directed graph, so there is a longest path from a triggering rule leaf to an end rule node which has no dependent join rules. The length of this path is the maximum number of iterations executed by the filter algorithm. Figure 9 shows an example filter run based on the tables presented in Section 3.3.4. The filter terminates with resource `doc.rdf#host` as result. After the filter terminated, all resources produced by end rules are transmitted to the appropriate LMRs.

3.5 Updates and Deletions

Updated and deleted resources can be determined by comparing the original RDF document with the updated, re-registered one. A resource is updated if it is contained in both documents, but at least one property is changed, added, or removed. A resource is deleted if it was contained in the original document but it is no more in the updated one. If a complete document is deleted all contained resources are deleted.

One execution of the MDV filter is not sufficient if updates and deletions are allowed. If a resource is updated three situations can be distinguished:

- The resource is no longer matched by a rule it previously was. It must be removed from an LMR's cache if this was the only rule the resource matched. If the resource still matches other rules it must stay in the cache.
- The resource is matched by a rule it previously was not. This situation is handled properly by the filter.
- The resource still matches all rules it previously had. All LMRs that cache this resource must update their cache to contain the modified resource.

Furthermore, resources referencing an updated or deleted resource must be considered. Assume the subscription rule:

`search` CycleProvider c `register` c
`where` c.serverInformation.memory $>$ 64

If a ServerInformation resource is updated, i.e., its memory property is set to 128, there can be CycleProvider resources that now match this rule. Whereas, if the ServerInformation resource's memory property is set to 32 or if the resource is deleted, there can be cached CycleProvider resources that must now be removed from the cache because they no longer match the rule, but only, if there are no other rules that the resource matches. Note that resources that are cached because of strong references are removed by the garbage collector, if necessary.

Our approach to solve all of this is to execute the filter multiple times, each time with different input data. First, the filter is executed with the original version of updated and deleted resources as input. The result is a set of so-called candidate resources. Each of these resources no longer matches at least one rule. We call

Initial Iteration

uri_reference	rule_id
doc.rdf#info	1
doc.rdf#info	2
doc.rdf#host	3

Iteration 1

uri_reference	rule_id
doc.rdf#info	4

Iteration 2

uri_reference	rule_id
doc.rdf#host	5

Figure 9. Table $ResultObjects$ for an Example Execution of the Filter

OID:	**search** CycleProvider c **register** c **where** c = *URI*
COMP:	**search** CycleProvider c **register** c **where** c.synthValue > *INT*
PATH:	**search** CycleProvider c **register** c **where** c.serverInformation.memory = *INT*
JOIN:	**search** CycleProvider c **register** c **where** c.serverHost **contains** 'uni-passau.de' **and** c.serverInformation.cpu = 600 **and** c.serverInformation.memory = *INT*

Figure 10. Benchmark Rule Types

them candidates because there can be other rules they still match or new rules they now match (after an update). Second, the modified metadata is written into the database and the filter is executed a second time, with the candidate resources as input. The result of the execution is a set of wrong candidate resources, that is, resources that must not be deleted. All true candidate resources (i.e., the set of resources determined in the first iteration excluding those obtained in the second iteration) can be deleted from LMRs' caches. Finally, the filter is executed a third time, now with the modified metadata as input. The last execution corresponds to the single filter execution that would suffice if no updates and deletions were allowed.

Alternatives to executing the filter multiple times are, e.g., to store for each resource a list of LMR's caching the resource. Or to use periodical cache invalidation, based on a time-to-live approach, resulting in resources dropping out of an LMR cache if they are not reinserted periodically.

4 Performance Experiments

Now, we present some performance experiments conducted using our prototype implementation of the filter algorithm. The results are important to decide if the filter should be started either when a new document is registered or periodically, to process several documents in one batch. All benchmarks were conducted on a Sun Enterprise 450 with 4GB memory running Solaris 2.7 and using Sun's Java JDK 1.2.2. As a backend we used a major commercial RDBMS.

We registered RDF documents similar to the document of Figure 1, each containing two resources, one of class CycleProvider, one of class ServerInformation. As rule base we used the rule types shown in Figure 10. For a single measurement, documents and rules of type OID, PATH, and JOIN were created in such a way that the CycleProvider resource in a document was matched by exactly one rule and that each rule matched exactly one resource (stored in one document) of all newly registered resources. COMP rules and corresponding documents were created in a way that a CycleProvider resource was matched by a certain percentage of the rules stored in the rule base, e.g., 10%. OID rules, which register a single CycleProvider resource with a given URI reference *URI*, and PATH rules, which register all CycleProvider resources with a synthValue property greater than the parameter *INT*, were both triggering rules. No decomposition was necessary and the filter algorithm did not need to evaluate any join rules. On the other hand, PATH and JOIN rules contained accesses to properties of referenced resources, so decomposition was necessary and join rules were created. Therefore, to evaluate them the complete filter algorithm was used.

We conducted the measurements with various batch sizes, an increasing rule base size, and different rule types. In a single measurement, we first created a rule base consisting of rules of the same type. Then, we registered a number of RDF documents and measured the overall runtime of the filter algorithm to process them. Depending on the rule type the algorithm terminated after the evaluation of all triggering rules (OID, COMP) or after the evaluation of all dependent join rules (PATH, JOIN). The average registration time of a single RDF document was calculated by dividing the overall runtime by the batch size. From the filter's point of view, registering several small documents and registering one large document is the same. So, these measurements also illustrate the behaviour of the filter algorithm regarding different document sizes.

Figures 11, 12, 13, and 14 show the dependency of the average registration costs from the batch size, i.e., the number of documents registered in one batch, for different rule types and rule base sizes.[5] For OID, PATH,

[5]Minor variations of the graphs are a consequence of the timing based on Java.

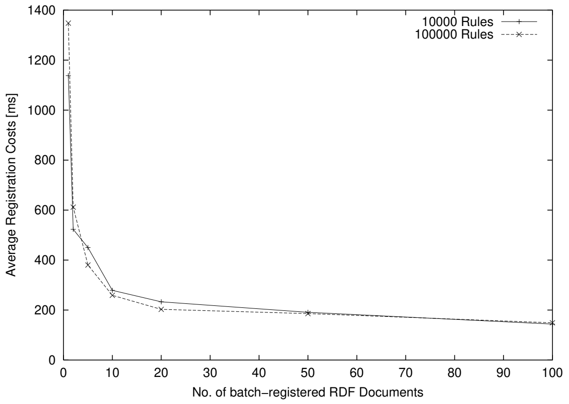


Figure 11. OID Rules

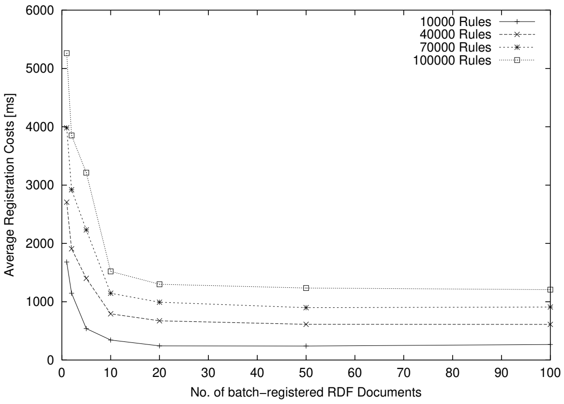


Figure 12. PATH Rules

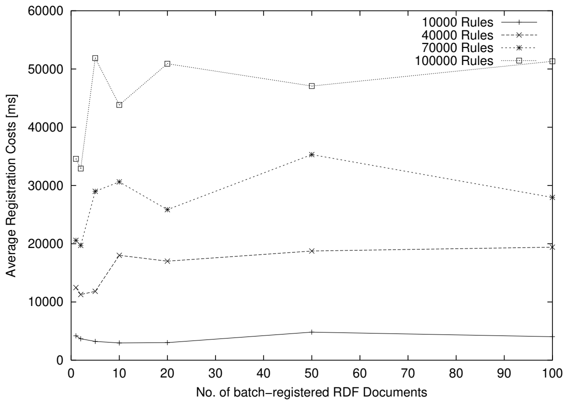


Figure 13. COMP Rules (10% of Rule Base)

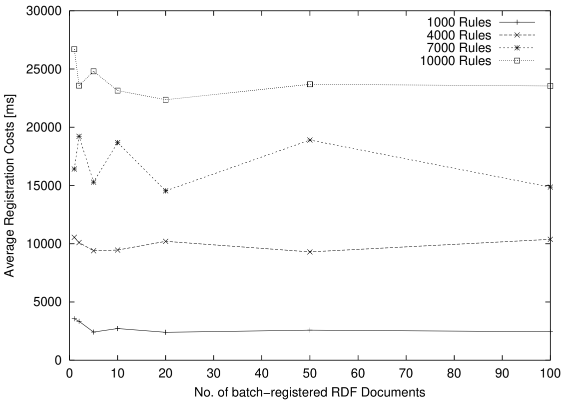


Figure 14. JOIN Rules

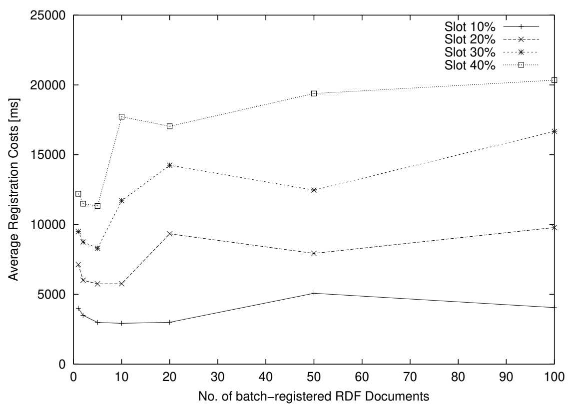
and JOIN rules the behaviour is basically the same. Registration of a small number of documents is more expensive than the registration of a lot of documents in one batch. From a certain batch size on (dependent on the rule type) the average registration costs are nearly constant. For COMP rules this is different; although the registrations costs are again nearly constant from a certain batch size on, registering few documents in one batch is preferable.

For simple OID rules the rule base size does not influence the runtime of the algorithm as the curves for 10,000 and 100,000 are almost identical. This is different for PATH, JOIN, and COMP rules where the registration costs do—as expected—depend on the rule base size, as Figures 12, 13, and 14 show.

Figure 15 shows the influence of the percentage of rules that match resources from the registered documents on the average registration costs. Not surprisingly a higher rule percentage results in higher registration costs independent of the batch size.

5 Related Work

Metadata management systems, data repositories, and catalogues are used in DBMSs since many years to store metadata about tables, indexes, and other data structures [30]. With the emergence of the Internet new metadata management systems with new challenges arose. UDDI [28] is a framework for storing and searching services provided by companies on the Internet. Contrary to MDV UDDI defines a fixed metadata schema and no automatic notification on data changes is provided. WebSemantics [23] searches the Internet for HTML pages that contain metadata about data sources and integrates them into a catalogue. The middleware system MOCHA [26] uses a (centralized) metadata repository to store Java classes and documentation about these classes in RDF. The Secure Service Discovery Service [11] stores metadata about network services in XML format. Lookup services like JINI [3], UPnP [29], and SLP [16] allow the discovery of plug-and-play services in networks but do not support large quantities of data or a query language. Some parts of our work are already described in [19].



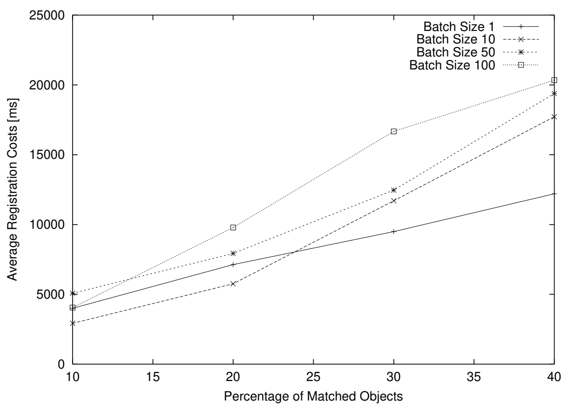


Figure 15. 10,000 COMP Rules - Varying Batch Sizes and Triggered Rule Base Percentage

Our filter algorithm uses triggering rules as an index to all subscription rules that are affected by new metadata. A similar approach is used in the publish & subscribe system Le Subscribe [25, 13]. It uses the predicates of subscription queries as index, but within the scope of a main memory algorithm. The Gryphon system [1] creates a tree from queries composed of simple predicates where each node represents a comparison. In [18] an Interval Binary Search Tree is introduced that stores intervals and allows to efficiently find all intervals that contain some given value. Whereas most publish & subscribe systems assume a distinguished data format, the information dissemination system SIFT [31] allows arbitrary (text) documents to be published. [2] presents a filter algorithm that uses XQL queries to select specific XML documents from an incoming stream of documents. To our knowledge none of these systems allows references between the information that is published, i.e., between different documents, as MDV does.

The NiagaraCQ [9] system evaluates queries continuously against a database. Queries are decomposed into partial queries, so that common partial queries are evaluated only once. A similar concept is implemented in OpenCQ [22] where queries are decomposed into events connected by the operators of the original query. Events are triggered by changes of the underlying data. Both systems and MDV can handle the insertion, update, and deletion of data. [27] introduced the concept of continuous queries but is restricted to append-only databases.

The cache of an LMR can be viewed as a set of materialized views with the corresponding rules as view definitions. [15, 4, 21] present some algorithms for updating materialized views. In [17] different strategies to update materialized views are investigated with respect to their performance. There are also similarities with semantic caching as described in [12].

6 Conclusion and Future Work

In this paper, we presented the architecture of MDV, a distributed metadata management system. MDV has a 3-tier architecture and supports caching and replication in the middle-tier so that information is stored near the users that need it and queries can be evaluated locally. By adding servers to (or removing servers from) the middle-tier as necessary our system can be adjusted easily to varying workloads. In order to keep replicas up-to-date and initiate the replication of new and relevant information, MDV implements a scalable publish & subscribe algorithm. We described this algorithm in detail, showed how it can be implemented using a standard relational database system, and presented the results of performance experiments conducted using our prototype implementation.

The work on the MDV system started about one year ago; its integration into the ObjectGlobe system was completed recently. For the future we are going to focus on the support for web services and their dynamic composition. This includes the utilization of XML as data format and XQuery as rule language—particularly within the publish & subscribe algorithm—as well as the support for such standards as UDDI [28] and WSDL [10] for the description, administration, and discovery of web services.

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 *Proc. of the ACM Symposium on Principles of Distributed Computing*, pages 53–61, 1999.

[2] M. Altinel and M. J. Franklin. Efficient Filtering of XML Documents for Selective Dissemination of Infor-

mation. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 53–64, 2000.

[3] K. Arnold. *The Jini(TM) Specification (The Jini(TM) Technology Series)*. Addison-Wesley, 1999.

[4] J. A. Blakeley, P.-Å. Larson, and F. W. Tompa. Efficiently Updating Materialized Views. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 61–71, 1986.

[5] R. Braumandl, M. Keidl, A. Kemper, D. Kossmann, A. Kreutz, S. Seltzsam, and K. Stocker. ObjectGlobe: Ubiquitous query processing on the Internet. *The VLDB Journal: Special Issue on E-Services*, 10(3):48–71, 2001.

[6] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler. Extensible Markup Language (XML) 1.0 (Second Edition). `http://www.w3.org/TR/REC-xml`, 2000.

[7] D. Brickley and R. V. Guha. Resource Description Framework (RDF) Schema Specification 1.0. `http://www.w3.org/TR/rdf-schema`, 2000.

[8] D. Chamberlin, D. Florescu, J. Robie, J. Siméon, and M. Stefanescu. XQuery: A Query Language for XML. `http://www.w3.org/TR/xquery`, 2001.

[9] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 379–390, 2000.

[10] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1. `http://www.w3.org/TR/wsdl`, 2001.

[11] S. Czerwinsky, B. Zhao, T. Hodes, A. Joseph, and R. H. Katz. An Architecture for a Secure Service Discovery Service. In *Proc. of ACM MOBICOM Conference*, pages 24–35, 1999.

[12] S. Dar, M. Franklin, B. Jónsson, D. Srivastava, and M. Tan. Semantic Data Caching and Replacement. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 330–341, 1996.

[13] 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 *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 115–126, 2001.

[14] D. Florescu and D. Kossmann. Storing and Querying XML Data Using an RDBMS. *IEEE Data Engeneering Bulletin*, 22(3):27–34, 1999.

[15] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining Views Incrementally. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 157–166, 1993.

[16] E. Guttman, C. Perkins, J. Veizades, and M. Day. Service Location Protocol, Version 2. `http://www.rfc-editor.org/rfc/rfc2608.txt`, 1999.

[17] E. Hanson. A Performance Analysis of View Materialization Strategies. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 440–453, 1987.

[18] E. N. Hanson, M. Chaabouni, C.-H. Kim, and Y.-W. Wang. A Predicate Matching Algorithm for Database Rule Systems. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 271–280, 1990.

[19] M. Keidl, A. Kreutz, A. Kemper, and D. Kossmann. Verteilte Metadatenverwaltung für die Anfragebearbeitung auf Internet-Datenquellen. In *Proc. GI Conf. on Database Systems for Office, Engineering, and Scientific Applications (BTW)*, Informatik aktuell, pages 107–126. Springer, 2001.

[20] O. Lassila and R. R. Swick. Resource Description Framework (RDF) Model and Syntax Specification. `http://www.w3.org/TR/REC-rdf-syntax`, 1999.

[21] B. G. Lindsay, L. M. Haas, C. Mohan, H. Pirahesh, and P. F. Wilms. A Snapshot Differential Refresh Algorithm. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 53–60, 1986.

[22] L. Liu, C. Pu, and W. Tang. Continual Queries for Internet Scale Event-Driven Information Delivery. *IEEE Transactions on Knowledge and Data Engineering*, 11(4):610–628, 1999.

[23] G. A. Mihaila, L. Raschid, and A. Tomasic. Equal Time for Data on the Internet with WebSemantics. In *Proc. of the Intl. Conf. on Extending Database Technology (EDBT)*, volume 1377 of *Lecture Notes in Computer Science*, pages 87–101. Springer, 1998.

[24] T. Özsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, 1999.

[25] J. Pereira, F. Fabret, F. Llirbat, and D. Shasha. Efficient Matching for Web-Based Publish/Subscribe Systems. In *Proc. of the Intl. Conf. on Cooperative Information Systems*, 2000.

[26] M. Rodriguez-Martinez and N. Roussopoulos. MOCHA: A Self-Extensible Database Middleware System for Distributed Data Sources. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 213–224, 2000.

[27] D. B. Terry, D. Goldberg, D. Nichols, and B. M. Oki. Continuous Queries over Append-Only Databases. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 321–330, 1992.

[28] Universal Description, Discovery and Integration (UDDI) Technical White Paper. Technical report, Ariba Inc., IBM Corp., and Microsoft Corp., 2000. `http://www.uddi.org`.

[29] Universal Plug and Play Device Architecture. Technical report, Microsoft Corporation, 2000. `http://www.upnp.org`.

[30] R. Williams, D. Daniels, L. Haas, G. Lapis, B. Lindsay, P. Ng, R. Obermarck, P. Selinger, A. Walker, P. Wilms, and R. Yost. R*: An Overview of the Architecture. In *Readings in Database Systems*, pages 515–536. Morgan Kaufmann Publishers, 1994.

[31] T. W. Yan and H. Garcia-Molina. The SIFT Information Dissemination System. *ACM Trans. on Database Systems*, 24(4):529–565, 1999.