Top *k* RDF Query Evaluation in Structured P2P Networks

Dominic Battré, Felix Heine, and Odej Kao

University of Paderborn, Fürstenallee 11, 33012 Paderborn, Germany,
{battre, fh, okao}@uni-paderborn.de,
WWW home page: http://www.upb.de/pc2

Abstract. Berners-Lee's vision of the Semantic Web describes the idea of providing machine readable and processable information using key technologies such as ontologies and automated reasoning in order to create intelligent agents.
The prospective amount of machine readable information available in the future will be large. Thus, heterogeneity and scalability will be central issues, rendering exhaustive searches and central storage of data infeasible. This paper presents a scalable peer-to-peer based approach to distributed querying of Semantic Web information that allows ordering of entries in result sets and limiting the size of result sets which is necessary to prevent results with millions of matches. The system relies on the graph-based W3C standard *Resource Description Framework* (RDF) for knowledge description. Thereby, it enables queries on large, distributed RDF graphs. [1]

1 Introduction

The Semantic Web [3] envisions to make the huge information resources of the Web available for machine-driven evaluation. Electronic agents are supposed to locate information necessary for their objectives, process them, generate conclusions and new information, and finally present the results to either a human user or to other electronic agents.

The Resource Description Framework (RDF, [11]) has been proposed by the W3C in order to formally describe resources. In combination with RDF Schema (RDFS, [4]) it provides sufficient expressibility to describe taxonomies of classes and properties and to infer information from taxonomies described with different schemas.

Query languages like SPARQL [15] with implementations like ARQ of Jena [2] (see http://esw.w3.org/topic/SparqlImplementations for other implementations) allow to query and infer information from RDF databases. These implementations, however, assume that all RDF triples are located in a central data repository, which is a questionable assumption given the growth of information available on the web.

[1] Partially supported by the EU within the 6th Framework Programme under contract 001907 "Dynamically Evolving, Large Scale Information Systems" (DELIS).

In [9] we have presented a scalable P2P based RDF querying strategy, which allows for distributed storage of information and selective collecting of RDF triples necessary to answer RDF queries. As current search engines of the web demonstrate, it is conceivable that RDF queries on real-world data will sometime deliver millions of results as well. Therefore, an evaluation procedure which retrieves every matching subgraph is neither desirable nor scalable. Typically, the user is not interested in an exhaustive collection of every matching result, but rather seeks for some matches which are good for her.

In this paper, we restrict to problem to finding the k best results (called *Top k* results) with respect to a single optimization criterion. I.e., the user might specify single variables in an `ORDER BY` clause of the query, but no complex expressions, and limit the number of results k by the `LIMIT` clause.

A simple solution would be to adapt the exhaustive search so that the results are ordered and filtered after the evaluation. However, a main goal of the Top k search is to enhance the scalability. Thus, we have to move away from the strategy to exhaustively collect all candidates before starting the final evaluation. We will rather start the final evaluation immediately, and fetch the candidates from the network step by step as needed. This avoids retrieving parts of the model graph which are never used during the query evaluation, when only k matches are of interest.

In the following section, we will provide an overview of the Top k query algorithm using an example. After that, section 3 describes the algorithm in detail and explains the caching strategies. The evaluation is given in section 4. Section 5 presents related work and section 6 concludes the paper.

2 Overview

In order to locate the RDF triples of various sources that are relevant to a query, we insert each triple three times into a distributed hash table (DHT) which is realized with Pastry [16]. Each triple is inserted into the hash map using the subject, the predicate, and the object as a key to the actual triple. That way, it is possible to look up all triples with a common subject for example and to perform the query processing by starting with one triple that has a URI in either subject, predicate, or object and to proceed from there on, fetching new triples and checking that their values to not contradict previous variable assignments.

Figure 1 shows an example query which consists of the following three triples: $t_1 = \langle v_1, U_1, v_2 \rangle$, $t_2 = \langle v_1, U_2, v_3 \rangle$, $t_3 = \langle v_3, v_1, v_2 \rangle$. U_1 and U_2 are fixed URIs, whereas v_1, v_2, and v_3 are variables. Assume that the user expects the value of v_2 to be a floating point value, and that she looks for matches with values of v_2 to be as large as possible.

As t_1 and t_2 have a bound value (the predicate) – a precondition for DHT lookups – either one can serve as a start for the query evaluation and we choose t_1 arbitrarily. By using U_1 as a DHT index and asking the responsible node to send triples with predicate U_1, we receive candidates for the variables v_1 and v_2. As the number of possible values of v_1 and v_2 can be very large, it is not

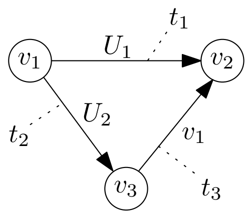


Fig. 1: Example RDF Query

desirable to fetch a complete list of matching triples. We rather need a way to query chunks of triples so that we can retrieve more candidates later, in case the first chunk did not deliver a sufficient number of matches. Thus, we ask the node to deliver the triples *in order*, so that we can later specify the highest known candidate to retrieve the next chunk. Because of scalability reasons, the target nodes do not store any state information. Therefore, the client node has to know the current chunk position and send it later to the other node to fetch the next chunk.

Assume the first chunk of five candidates gets retrieved and stored in a table as depicted in figure 2a. By selecting the first candidate, v_1 gets bound to A and v_2 to 10. As t_2 has a bound subject and predicate now, we can choose this as the next triple to proceed recursively. Here, we have to fetch candidates which respect the current variable bindings. As the predicate is bound to U_2, we can use U_2 as DHT index, and retrieve all triples with predicate U_2 which have the subject A. We sort the results by ascending order of v_3.

Assume that the result are two triples, $\langle A, U_2, X\rangle$ and $\langle A, U_2, Y\rangle$. These triples are stored as candidates for t_2 and the first one is selected in the backtracking procedure.

t_1

v_1	U_1	v_2
A	U_1	10
C	U_1	9
B	U_1	8.5
A	U_1	7
B	U_1	7

t_2

v_1	U_2	v_3

t_3

v_3	v_1	v_2

(a) Candidate Lists.

t_1

v_1	U_1	v_2
A	U_1	10
C	U_1	9
B	U_1	8.5
A	U_1	7
B	U_1	7

t_2

v_1	U_2	v_3
A	U_2	X
A	U_2	Y

t_3

v_3	v_1	v_2
X	A	10

(b) First match.

Fig. 2: Query evaluation

Finally, we fetch candidates for the last triple. As it consists only of variables, we have to use the current binding of one of the variables as DHT index. We choose v_1, and thus use A as DHT index. The remaining two variables are already bound to $v_3 = X$ and $v_2 = 10$. That means that we ask for the *existence* of the triple $\langle X, A, 10\rangle$. As the triple exists, we have generated the first match (see figure 2b). Afterwards, we backtrack to t_2, select the second candidate, and ask for the existence of $\langle Y, A, 10\rangle$ (see figure 3). By this procedure, we generate the top matches step by step, only retrieving the candidates as needed. After having generated the requested number of matches, the procedure stops.

However, as we can see e.g. from the last step, it might be useful to have a kind of look-ahead when fetching the candidates. We have contacted the node for URI A twice in short succession to ask for the existence of triple $\langle X, A, 10\rangle$ and then $\langle Y, A, 10\rangle$. During the first lookup, the second candidate for t_2 was already known, and therefore, it should have been possible to ask directly for the existence of the second triple in order to save one communication step.

t_1

v_1	U_1	v_2
A	U_1	10
C	U_1	9
B	U_1	8.5
A	U_1	7
B	U_1	7

t_2

v_1	U_2	v_3
A	U_2	X
A	U_2	Y

t_3

v_3	v_1	v_2
X	A	10
Y	A	10

Fig. 3: Second match

3 Top *k* algorithm and caching strategy

In this section we present an evaluation strategy with look-ahead caches that efficiently reduce the amount of information transferred over the network and the number of messages passed between nodes.

The evaluation function, as we can see in figure 4, resembles the basic backtracking strategy as described in the previous section. Its parameters *nr_matches* and k specify the number of matches found already and the number of matches to be delivered in total respectively. The ordered list of triples (T_i) describes the query and i represents the recursion depth, as query triples are matched to RDF graph triples with backtracking. We follow the notation of [9] by denoting with $\mathcal{L}$ the set of labels (XML literals and URI references) and with $\mathcal{B}$ the set of blank nodes. With this notation, we can describe the binding of variables $\{v_i\}$ to their actual nodes in the RDF graph as a partial function

$$B : \{v_i\} \to \mathcal{L} \cup \mathcal{B}. \tag{1}$$

Similarly, we maintain a set of candidates that can possibly be assigned to variables, with

$$C : \{v_i\} \to \mathrm{Pow}(\mathcal{L} \cup \mathcal{B}). \tag{2}$$

```
function eval($nr\_matches$, $k$, $(T_i)$, $i$, $B$, $C$, $Caches$)
  if $i = |(T_i)| + 1$ then
    record $B$ as a match found; /* all triples are bound */
    return $nr\_matches + 1$;
  end if
  $j := 0$; /* counter of inspected candidate triples for $T_i$ */
  loop
    $t := Caches$.getCache($T_i$).getNextCandidate($B$, $C$, $j$);
      /* the candidate will respect the bindings in $B$ */
    $j := j + 1$;
    if $t = null$ then
      break; /* no more candidates available */
    end if
    /* update bindings and candidates: */
    $B' := B \cup$ Bindings of $t$;
    $C' := C \cup$ Candidates of $t$;
    $nr\_matches :=$ eval($nr\_matches$, $k$, $(T_i)$, $i + 1$, $B'$, $C'$, $Caches$);
    if $nr\_matches \geq k$ then
      break;
    end if
  end loop
  return $nr\_matches$;
end function
```

Fig. 4: Evaluation algorithm

Finally, we define a set of caches that allow retrieving candidates for the RDF query triples. Each triple in the query has one individual cache, but we employ different kinds of caches as we will see later on.

The general idea of the evaluation function is to iterate over all possible assignments to triples (variable j serves as an index for the iteration), assume one, and proceed to a recursive evaluation until we encounter contradictions, find a complete match, realize that we have found a sufficient number of matches, or until we cannot assign any more triples. The caches allow to perform this search efficiently even though data are distributed among the peers of the network.

We will describe the strategy of the caches with the example of fetching candidates for triple $t_3 = (v_3, v_1, v_2)$ in figure 2b. As described before, we use the predicate $v_1 = A$ as the DHT key and have two remaining components of the triple to look up; v_3 and v_2. These variables are already bound to the valued $v_3 = X$ and $v_2 = 10$. If the cache knows whether the resulting triple $\langle X, A, 10\rangle$ exists or does not exist, it can return this answer. Otherwise it has to retrieve the information of the node that is in charge of triples with predicate A. Instead of fetching just one triple it queries this and a chunk of additional queries, anticipating that they will be requested later. As we know from the first occurrences of v_1 and v_3 in columns of figure 2b, the candidates of v_1 are $\{A, C, B\}$, and the candidates of v_3 are $\{X, Y\}$. The cross product of both candidate sets defines a super set of values of possible interest. The cache asks not only whether the triple $\langle X, A, 10\rangle$ exists but also asks for additional $c - 1$ unknown triples of the cross product, where c is the chunk size. That way, we hope to retrieve information that will be requested later.

Each triple can be split into one component which defines the key of the DHT and two remaining components. If possible, we choose a fixed URI as DHT key, otherwise we iterate over all possible candidates of a variable. In the previous example, the predicate served as DHT key, and subject and object served as the remaining components. The latter ones were both bound variables, but this does not need to be the case. In general we can encounter six different cases, where the two remaining components are:

1. two unbound variables,
2. an unbound variable plus a bound variable,
3. an unbound variable plus an fixed URI or literal,
4. two bound variables,
5. a bound variable plus an fixed URI or literal, and
6. two URIs / literals

A component is bound if it consists of a variable that was seen before in a higher recursion level, unbound with a variable that occurs for the first time, or fixed if it is a URI or literal. The binding of a variable is the known candidate set, where the variable was found the first time. For each of these cases we define a specially optimized type of cache. These caches are depicted in figure 5.

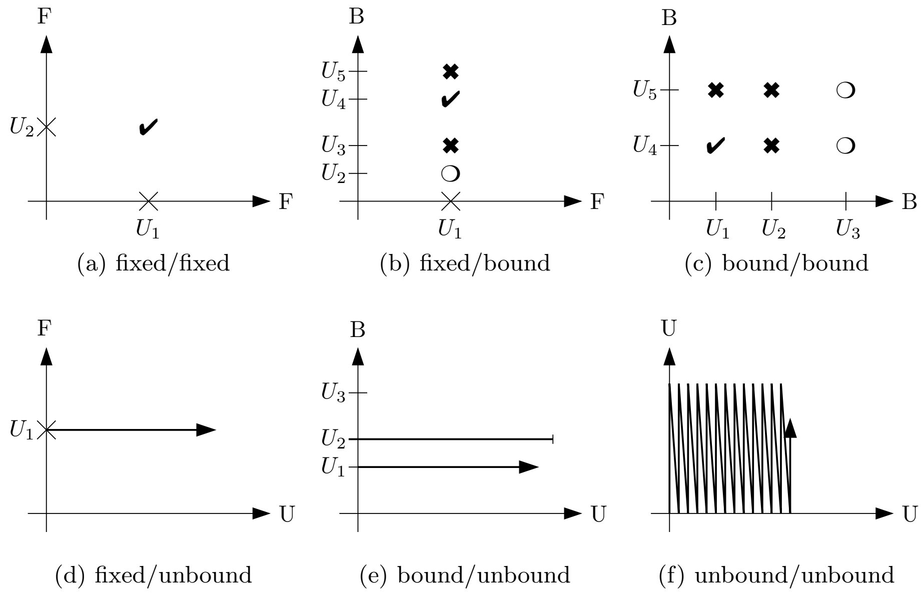


Fig. 5: Cache Types

The caches have to query the next chunk of up to c triples for a query. They deliver these chunks triple by triple. For scalability reasons, the peer who will process the query, does not store any state information, so the requesting peer is in charge of submitting the state along with the actual request. The state can consist of the set of triples we want to gather information about (first three cases below) or of a set of markers, which define the last triples for which we know information already (last three cases below).

The simplest cache for fixed/fixed components (see fig. 5a), which occur if a RDF query contains a triple with three URIs, does a simple lookup without look-ahead. The state of the cache can be "triple exists in RDF graph" (represented by a check mark in the figure), "triple does not exist in RDF graph" (cross), or "unknown whether triple exists in RDF graph" (circle).

For fixed/bound component pairs (see fig. 5b) the caching is simple as well. A peer requests a chunk of triples by specifying the fixed component and a set of candidates for the bound component for which it wants to retrieve the state.

For bound/bound components (see fig. 5c) we build up a request containing a set of unknown combinations of already known values for the bound variables.

The fixed/unbound cache (see fig. 5d) is similar to the fixed/bound cache, except that it is sufficient to request the next c elements starting after a given position. Therefore, we submit the fixed element and the last inspected value for the unbound element as the request.

The bound/unbound cache (see fig. 5e) extends this by storing and submitting markers for the last known elements in several rows. The peer who processes a request starts sending triples at the first marker until c triples have been sent or continues at the next marker if the row (candidates) do not provide c triples.

For the unbound/unbound cache (see fig. 5f) it is again sufficient to submit a single marker which determines the next triples to be delivered.

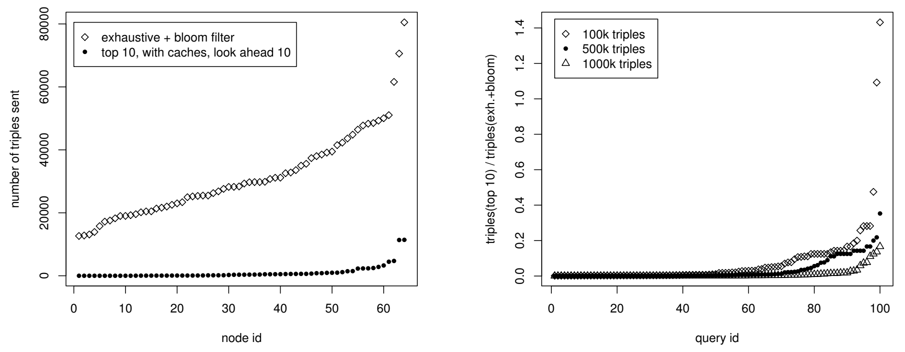
4 Evalution

For the evaluation of the strategy described we have generated random resource descriptions that follow the data guide of the JSDL specification [7]. The generation was based on rough but arbitrary estimations (e.g.: of the many operating systems available, the first three will account for the majority of offers and requests). The data generation approach is very similar to the Lehigh University Benchmark (LUBM), see for example [8]. The resource descriptions consist of 11.3 RDF triples on average (standard deviation: 2.6). Queries are less specific than resource descriptions and consist of 3.9 RDF triples on average (s.d.: 1.8). This ensures big result sets which are focus of the Top k strategy.

We have evaluated the Top k strategy for $k = 10$ with a look-ahead of 10 triples against an optimized exhaustive search described in [9]. This exhaustive search employs sophisticated means based on Bloom filters to fetch a minimal set of triples necessary to do a full evaluation locally. For the evaluation, we have processed 100 queries on databases of 100,000, 500,000, and 1,000,000 RDF triples, spread on a P2P network of 64 nodes.

Our main goals were to reduce the number of triples sent over the network and to reduce the number of messages sent over the network.

Figure 6a shows the aggregated number of queries each individual peer had to sent to process all 100 queries on the database of 100,000 triples. We see that some nodes were not involved at all, when trying to find just the first 10 matches. On average, each peer sent 634.7 triples in 308.2 messages for a Top 10 evaluation, while the exhaustive evaluation sent an average of 31485 triples in 18886 messages per node.



(a) Total number of triples sent by peers to process exhaustive and Top 10 search

(b) Ratio of triples sent by peers for Top 10 search divided by triples sent for exhaustive search

Fig. 6: Empirical analysis

We further hypothesize that restricting the number of results gets increasingly important the larger the database is. Therefore, we have analyzed the ratio of triples sent over the network with Top k strategy devided by the number of triples sent with optimized exhaustive search for all three databases. In two of one hundred test queries, the exhaustive search was slightly faster than the Top k search on 100,000 triples because the triple ordering strategy in the Top k search is less optimized. These two queries had very small result sets. In all 98 other test queries, the Top k strategy was superior. In order to disregard these outliers, we describe the median values for the ratios instead or the mean values.

The experimental results support our hypothesis (see fig. 6b). The median ratio of triples sent over the network for Top k divided by the the number of triples sent for an exhaustive search of 100 queries was just 0.89% for a database of 100,000 triples, 0.13% for 500,000 triples, and 0.05% for 1,000,000 triples. The mean ratios were 5.7%, 2.8%, and 0.87% respectively. We see that the Top k is on average significantly faster than the optimized exhaustive search if result sets are big.

Decreasing the look-ahead from 10 triples to 1 increases the total number of triples sent over the network by a factor of 1.61 and the total number of messages by a factor of 1.95 on the database of 100,000 triples.

5 Related Work

The general idea of the semantic web [3] paints the vision of a web where information can be automatically processed by software. The Resource Description Framework (RDF) together with RDF Schema [11, 4] is one of the upcoming standards which will help to make this vision a reality.

Kokkinidis and Christophides describe in [12] a P2P based middleware for evaluating queries in the RDF Query Language (RQL) using RDF Schema knowledge. They focus on the construction and optimization of query plans. Their basic approach is different from ours as they require mandatory schema information encoded in RDFS. In our approach, schema information is not required for query processing.

In [5], dynamic query execution for schema-based P2P networks in the context of the Edutella project [14] is described. This work focuses on dynamic query planning and execution. Queries are evaluated in a distributed fashion; the optimizer tries to evaluate operators local to the data.

Kokkinidis et al. do not address Top k evaluation explicity. Nejdl et al. propose a Top k evaluation strategy in [13] but this is fundamentally different from our approach as it is based on a P2P network with super-peer architecture.

The idea of using URIs as the key to distribute information over an DHT-based P2P network has been described in several papers. We have used it in [10] to distribute knowledge based on Description Logics and it has been used in BabelPeers [9], the GridVine project [1], and RDFPeers [6] to distribute RDF triples. The distribution of triples used in this paper is similar to these ideas.

6 Conclusion

In this paper, we have focussed on querying large amounts of distributed RDF-based knowledge. While the Semantic Web is a prominent use case for our algorithm, we argue that other applications like Grid resource discovery are important as well. In most use cases, only a small fraction of the results are relevant for the user. Thus, we devised a Top k query algorithm which delivers only the k best results according to a sorting attribute. The algorithm operates on RDF data distributed over an DHT-based P2P network. It uses caching and look-ahead strategies to reduce both the number of messages and their size.

In the evaluation, we showed that the algorithm indeed reduces network usage significantly compared to an exhaustive evaluation. We further showed that the positive effect increases the larger the underlying RDF knowledge-base grows. Thus our strategy is efficiently increasing scalability of RDF-based P2P data management systems.

References

1. Karl Aberer, Philippe Cudré-Mauroux, Manfred Hauswirth, and Tim Van Pelt. GridVine: Building Internet-Scale Semantic Overlay Networks. In Sheila A. McIlraith, Dimitris Plexousakis, and Frank van Harmelen, editors, *International Semantic Web Conference*, volume 3298 of *Lecture Notes in Computer Science*, pages 107–121. Springer, 2004.
2. ARQ - A SPARQL Processor for Jena. URL `http://jena.sourceforge.net/ARQ/`.
3. Tim Berners-Lee, James Hendler, and Ora Lassila. The Semantic Web. *Scientific American*, May 2001.
4. Dan Brickley and Ramanathan V. Guha. RDF Vocabulary Description Language 1.0: RDF Schema. `http://www.w3.org/TR/rdf-schema`, 2004.
5. Ingo Brunkhorst, Hadhami Dhraief, Alfons Kemper, Wolfgang Nejdl, and Christian Wiesner. Distributed Queries and Query Optimization in Schema-Based P2P-Systems. In *Databases, Information Systems, and Peer-to-Peer Computing, First International Workshop, DBISP2P, Berlin Germany, September 7-8, 2003, Revised Papers*, pages 184–199, 2003.
6. Min Cai, Martin Frank, Baoshi Pan, and Robert MacGregor. A Subscribable Peer-to-Peer RDF Repository for Distributed Metadata Management. *Journal of Web Semantics: Science, Services and Agents on the World Wide Web*, 2(2), 2005.
7. Andreas Savva (editor). Job Submission Description Language (JSDL) Specification, Version 1.0, 2005.
8. Yuanbo Guo, Jeff Heflin, and Zhengxiang Pan. Benchmarking DAML+OIL Repositories. In *International Semantic Web Conference*, pages 613–627, 2003.
9. Felix Heine. Scalable P2P based RDF Querying. In *First International Conference on Scalable Information Systems (IN FOSCALE06), to appear*, 2006.
10. Felix Heine, Matthias Hovestadt, and Odej Kao. Towards Ontology-Driven P2P Grid Resource Discovery. In *GRID*, pages 76–83. IEEE Computer Society, 2004.
11. Graham Klyne and Jeremy J. Carroll. Resource Description Framework (RDF): Concepts and Abstract Syntax. `http://www.w3.org/TR/rdf-concepts`, 2004.
12. Giorgos Kokkinidis and Vassilis Christophides. Semantic Query Routing and Processing in P2P Database Systems: The ICS-FORTH SQPeer Middleware. In *Current Trends in Database Technology - EDBT 2004 Workshops, EDBT 2004 Workshops PhD, DataX, PIM, P2P&DB, and ClustWeb, Heraklion, Crete, Greece, March 14-18, 2004, Revised Selected Papers*, pages 486–495, 2004.
13. Wolfgang Nejdl, Wolf Siberski, Uwe Thaden, and Wolf-Tilo Balke. Top-k Query Evaluation for Schema-Based Peer-to-Peer Networks. In Sheila A. McIlraith, Dimitris Plexousakis, and Frank van Harmelen, editors, *ISWC 2004: Third International Semantic Web Conference*, volume 3298, pages 137–151, jan 2004.
14. Wolfgang Nejdl, Boris Wolf, Changtao Qu, Stefan Decker, Michael Sintek, Ambjörn Naeve, Mikael Nilsson, Matthias Palmér, and Tore Risch. EDUTELLA: a P2P networking infrastructure based on RDF. In *WWW2002, May 7-11, 2002, Honolulu, Hawaii, USA*, pages 604–615, 2002.
15. Eric Prud'hommeaux and Andy Seaborne (Editors). SPARQL Query Language for RDF. URL `http://www.w3.org/TR/rdf-sparql-query/`, Nov 2005.
16. Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Proc. of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, 2001.