ViP2P: Efficient XML Management in DHT Networks

Konstantinos Karanasos[1], Asterios Katsifodimos[1],
Ioana Manolescu[1], and Spyros Zoupanos[2]

[1] Inria Saclay–Île de France and LRI, Université Paris Sud-11
[2] Max-Planck Institut für Informatik, Saarbrücken, Germany
fistname.lastname@inria.fr

Abstract. We consider the problem of efficiently sharing large volumes of XML data based on distributed hash table overlay networks. Over the last three years, we have built ViP2P (standing for *Vi*ews in *P*eer-to-*P*eer), a platform for the distributed, parallel dissemination of XML data among peers. At the core of ViP2P stand *distributed materialized XML views*, defined as XML queries, filled in with data published anywhere in the network, and exploited to efficiently answer queries issued by any network peer. ViP2P is one of the very few *fully implemented* P2P platforms for XML sharing, deployed on *hundreds of peers in a WAN*. This paper describes the system architecture and modules, and the engineering lessons learned. We show experimental results, showing that our choices, outperf related systems by orders of magnitude in terms of data volumes, network size and data dissemination throughput.

Keywords: P2P, XML, DHT, distributed views

1 Introduction

We consider the large-scale management of distributed XML data in a *peer-to-peer* (P2P) setting. To provide users with *precise and complete* answers to their requests for information, we assume that the requests are formulated by means of a structured query language, and that the system must return complete results. That is, if somewhere in the distributed peer network, an answer to a given query exists, the system will find it and include it in the query result. Thus, we consider P2P XML data management based on a structured peer-to-peer network, more specifically, a distributed hash table (or DHT, in short).

In this setting, users may formulate two kinds of information requests. First, they may want to *subscribe to interesting data anywhere in the network*, that were published before or after the subscription is recorded in the system. We need to ensure that results are eventually returned as soon as possible, following the publication of a matching data source. Second, users may formulate *ad-hoc (snapshot)* queries, by which they just seek to obtain as fast as possible the results which have already been published in the network.

The challenges raised by a DHT-based XML data management platform are: (*i*) building a *distributed resource catalog*, enabling data producers and consumers to "meet" in an information sharing space; such a catalog is needed both

for subscription and ad-hoc queries; (*ii*) efficiently distributing the data of the network to consumers that have subscribed to it, and; (*iii*) providing *efficient distributed query evaluation algorithms* for answering ad-hoc queries fast.

Over the last three years, we have invested more than 6 man-years building the ViP2P[3] platform to address these challenges. Importantly, ViP2P uses *subscriptions as views*: results of long-running subscription queries are stored by the subscriber peers and re-used to answer subsequent ad-hoc queries.

A critical engineering issue when deploying XML data management applications on a DHT is the division of tasks between the DHT and the upper layers. In ViP2P, we chose to load the DHT layer *as little as possible*, and keep the heavyweight query processing operations in the data management layer and outside the DHT. This has enabled us to build and efficiently deploy an important-size system (70.000 lines of Java code), which we show scales on up to 250 computers in a WAN, and hundreds of GBs of XML data.

Several DHT-based XML data management platforms [3,9,11] are only concerned with *locating* in the P2P networks the documents relevant for a query. All the peers which may hold results then locally evaluate the query, leading to high query traffic and peer overload. In contrast, as in [1,2,6,8], ViP2P answers queries over a global XML database distributed in a P2P network.

In this paper, we make the following contributions. (*i*) We present a scalable, end-to-end architecture of one of the very few DHT-based XML sharing platforms actually implemented. From a system engineering perspective, we believe this is a useful addition to the corpus of existing DHT-based XML data management literature which has focused more on indexing and filtering algorithms, and less on system aspects. (*ii*) We present an experimental study of XML dissemination to DHT-based subscriptions, show which network parameters impact its performance, and demonstrate that ViP2P outperforms competitor systems by orders of magnitude in terms of published data volumes and throughput.

In the sequel, Section 2 presents an overview of the platform and Section 3 the architecture of a ViP2P peer. Section 4 experimentally demonstrate the platform scalability; many more experiments can be found in [4]. We then conclude.

2 Platform Overview

XML data flows in ViP2P can be summarized as follows. XML documents are *published* independently and autonomously by any peer. Peers can also formulate *subscriptions*, or long-running queries, potentially matching documents published before, or after the subscriptions. The results of each subscription query are *stored* at the peer defining the subscription, and the definition of it is *indexed* in the peer network. Finally, peers can ask *ad-hoc queries*, which are answered in a snapshot fashion (based on the data available in the network so far) by exploiting the existing subscriptions, which can be seen as *materialized views*. In this Section, we detail the overall process via an example.

A sample ViP2P instance over six peers is depicted in Figure 1 (left). In the Figure, XML documents are denoted by triangles, whereas views are denoted by tables, hinting to the fact that they contain sets of tuples.

[3] http://vip2p.saclay.inria.fr

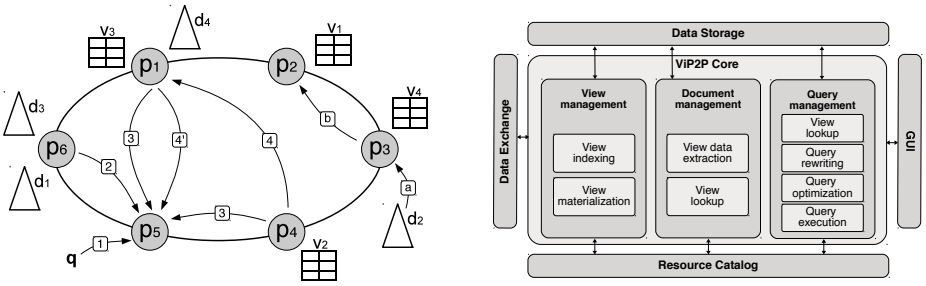

Fig. 1. System overview (left); Architecture of a ViP2P peer (right).

For ease of explanation, we make the following naming conventions for the remainder of this paper: *publisher* is a peer which publishes an XML document, *consumer* is a peer which defines a subscription and stores its results (or, equivalently, the respective materialized view) and *query peer* is a peer which poses an *ad-hoc* query. Clearly, a peer can play any subset of these roles simultaneously.

View publication A ViP2P view is a *long-running subscription query* that any peer can freely define. In the sequel, we will refer to long-running subscription queries as *materialized views* or just *views*. The *definition* (i.e., the actual query) of each newly created view is indexed in the DHT network. For instance, assume peer p_2 in Figure 1 (left) publishes the view v_1, defined by the XPath query $//bibliography//book[contains(.,'Databases')]$. The view requires all the books items from a bibliography containing the word 'Databases'. ViP2P indexes v_1 by inserting in the DHT the following three (key, value) pairs: $(bibliography, v_1@p_2)$, $(book, v_1@p_2)$ and $('Databases', v_1@p_2)$. Here, $v_1@p_2$ encapsulates the structured query defining v_1, and a pointer to the concrete database at peer p_2 where v_1 data is stored. As will be shown below, all existing and future documents that can affect v_1, *push* the corresponding data to its database.

Peers look up views in the DHT in two situations: when publishing documents, and when issuing ad-hoc queries. We detail this below.

Document publication When publishing a document, each peer is in charge of identifying the views within the whole network to which its document may contribute. For instance, in Figure 1 left (step a), peer p_3 publishes the document d_2. Peer p_3 extracts from d_2 all distinct element names and all keywords. For each such element name or keyword k, p_3 looks up in the DHT for view definitions associated to k. Assume that document d_2 contains data matching the view v_1 as it contains the element names *bibliography* and *book*, as well as the word $'Databases'$, thus p_3, learns about the existence of v_1 (step b). In the publication example above, p_3 extracts from d_2 the results matching v_1; from now on, we will use the notation $v_1(d_2)$ to designate such results. Peer p_3 sends $v_1(d_2)$ to p_2 (step c), which adds them to the database storing v_1 data.

Ad-hoc query answering ViP2P peers may pose *ad-hoc queries*, which must be evaluated immediately. To evaluate such queries, a ViP2P peer looks up in the network for views which may be used to answer it. For instance, assume the query $q = //bibliography//book[contains(.,'Databases')]//author$ is issued at peer p_5

(step 1, in Figure 1, left). To process q, p_5 looks up the keys *bibliography*, *book*, $'Databases'$ and *author* in the DHT, and retrieves a set of view definitions, v_1, v_2 and v_3 (step 2). Observe that q can be rewritten as $v_1//author$; therefore, p_5 can answer q just by retrieving and extracting q's results out of v_1. Alternatively, assume that q can also be rewritten by joining views v_2 and v_3 as ViP2P can combine *several* views to rewrite a query [4]. In that case, p_5 can retrieve the views v_2 and v_3 (step 3) and join them to evaluate q. However, the whole content of both views has to be shipped to p_5 to evaluate the query q. Instead, v_2 can be shipped to peer p_1, joined locally with v_3 at p_1 (step 4), who will send the query results to the query peer (step 4'), avoiding extraneous data transfers.

3 ViP2P Peer Architecture

Figure 1 (right) outlines the architecture of a ViP2P peer. In this Section, we introduce the auxiliary modules on which every peer relies, and then move to the main modules, which are included in the *ViP2P Core* box of Figure 1.

Resource catalog provides the underlying DHT layer used to keep peers connected, and to index and lookup views. It employs the FreePastry DHT, which is an open-source implementation of the Pastry overlay network [10]. It provides efficient request routing, deterministic object location, and load balancing.

Data exchange module is responsible *for all data transfers* and relies on Java RMI. Experience with FreePastry has shown that Pastry-routed inter-peer communications quickly become the bottleneck when sending important volumes of data [1]. Instead, we use RMI (with our own (de)serialization methods, properly controlling concurrency at the sender and receiver side etc.) to send larger messages containing view tuples, when views are materialized and queried.

Data storage Within each peer, view tuples are efficiently stored into a native store that we built using the BerkeleyDB[4] library. It allows storing, retrieving and sorting entries, with transactional guarantees for concurrent operations.

The VIP2P GUI enables publishing views, documents and evaluating queries.

We now describe the core modules.

Document management determines to which views the peer's documents may contribute data, and extracts and sends this data to the appropriate consumers.

- View definition lookup When a new document is published by a peer, this module looks up in the DHT for view definitions to which the document may contribute data. The result is a superset of view definitions of the views that the document might contribute data to. These definitions are then passed to the *view data extraction* module.

- View data extraction Given a list of view definitions, this module at a publisher peer extracts from the document the tuples matching each view, and ships them, in a parallel fashion, to the corresponding consumers. The view data extractor is capable of simultaneously matching several views on a given document, thus extracting the corresponding tuples at a single document traversal.

[4] http://www.oracle.com/technetwork/database/berkeleydb/

View management This module handles view indexing and materialization.

- *View indexing* This module implements the view indexing process. In this context, a given algorithm for extracting (key, value) pairs out of a view definition is termed a *view indexing strategy* [4]. In our experiments, the most efficient is the *Label Indexing (LI)* strategy, indexing a view v by each v node label (element or attribute name, or word).

- View materialization The *view materialization* module receives tuples from remote publishers and stores them in the respective BerkeleyDB database. In a large scale, real-world scenario, thousands of documents might be contributing data to a single view. To avoid overload on its incoming data transfers, this module implements a back-pressure *tuple-send/receive protocol* which informs the publisher when the consumer is overloaded, so that the publisher waits until the consumer is ready to accept new tuples.

Query management comprises the following modules for query evaluation.

- *View lookup* This module, given a query, performs a lookup in the DHT network retrieving the view definitions that may be used to rewrite the query. Depending on the indexing strategy (mentioned earlier in this Section), this module uses a different *view lookup* method.

- *Query rewriting* Given a query and a set of view definitions, this module produces a logical plan which, evaluated on some views, produces exactly the results required by the query (algorithm detailed in [7]).

- *Query optimization* This module receives a logical plan that is output by the *query rewriting* module, and translates it to an optimized physical plan. The optimization concerns both the logical (join reordering, push selections/projections etc.) and physical (dictating the exact flow of data during query execution) level.

- *Query execution* This module provides a set of physical operators which can be executed by any ViP2P peer, implementing the standard iterator-based execution model. Since ViP2P is a distributed application, operators can be deployed to peers and executed in a distributed manner. The query optimization module is the one to decide the parts of a physical plan that every peer executes.

4 Experimental Results

We now present a set of experiments studying ViP2P performance, carried on the Grid5000 infrastructure[5]. Due to space limitations, we only report here on our main findings; many more experiments are described in [4].

In our experiments, we used synthetic "product catalog" documents of controllable size (more details can be found in [4]). First, all views are created and indexed. Then, on a signal sent to all publishers, these peers start publishing all their documents as fast as possible. This is a "flash crowd" scenario, aiming at stress-testing our system. Queries are posed and processed after all the views are filled with data. Section 4.1 examines view materialization, while Section 4.2 studies the performance of the query execution engine.

[5] https://www.grid5000.fr

4.1 View Materialization in Large Networks

We present three materialization experiments; many more can be found in [4].

Experiment 1: one publisher, varying data size, 64 consumers In this experiment we study how materialization time is affected when the total size of published data is increased. We use one publisher holding all the data in the network. The size of the published data varies from 64MBs to 1024MBs.

Each of the 64 consumers holds one view of the form $//catalog//camera_{K\ cont}$ where K varies according to the peer that holds the view. For example, the first consumer holds the view $//catalog//camera_{1\ cont}$, the second holds the view $//catalog//camera_{2\ cont}$ etc. This way, from each document the publisher extracts 64 tuples, each of which is sent to a different consumer. All the content of the documents is absorbed by the 64 views.

We run two variations of the same experiment: (i) one for sequential tuple sending where a publisher sends the tuples to their corresponding consumers one after the other, and (ii) one for parallel tuple sending, where a publisher ships the tuples to their corresponding consumers simultaneously. The graph at left in Figure 2 shows, as expected, that the materialization time increases linearly with the size of data published in the network in both cases. It also shows that the materialization time in the case of parallel tuple sending is considerably shorter (about 3000 sec. instead of 11500 sec. for absorbing 1024MBs of data).

Experiment 2: 64 publishers, varying data size, one consumer We now focus on the impact of the number of (simultaneous) publishers on the capacity of absorbing the data into a single view. The published data size varies from 64MBs to 3.2GBs, and is equally distributed to 64 publishers. All the published data ends up in one view. Similarly to Experiment 1, we test 2 modes of tuple-receiving concurrency: (i) sequential tuple receiving and; (ii) parallel tuple receiving.

Figure 2 (center) depicts the materialization time as the size of the published data increases. We observe that the materialization time increases proportionally to the size of published data in both sequential and parallel tuple receiving modes. Also, parallel tuple receiving reduces the view materialization time by more than 50% (600 sec. instead of about 1400 sec. to absorb 3.2GBs of data).

From the two graphs (left, center) in Figure 2, we conclude that it is faster for the network to absorb data using one consumer and many publishers rather than many consumers and one publisher since it is slow for a peer to extract all the available data by itself and ship them to the consumers.

Experiment 3: Community publishing A "community publishing" setting is the closest to real world scenarios: a large and complex environment, with many publishers and many consumers. We use a network of 250 peers, each of which holds the same number of 1MB documents. We logically divide the network into 50 groups of 5 publishers and one consumer each. The data of all publishers in a group is of interest *only* to the consumer of that group.The total amount of data published (and shipped to the views) varies from 20GBs to 160GBs.

Figure 2 (right) shows that the materialization time grows linearly with the published data size. This experiment demonstrates the good scalability properties of ViP2P as the data volume increases. Moreover, it shows that ViP2P

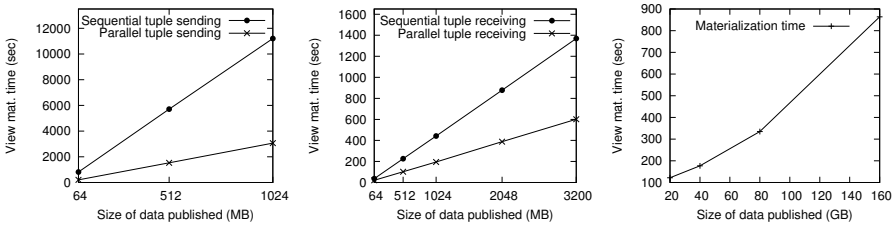

Fig. 2. Experiment 1: one publisher, varying size of data, 64 consumers (left); Experiment 2: 64 publishers, varying data size, one consumer (center); Experiment 3: publishing varying size of data in 50 groups of 5 peers each (right).

exploits many parallelization opportunities in such "community publishing" scenarios when extracting, sending, receiving and storing view tuples. Here we report on sharing up to 160 GB of data over up to 250 peers with a throughput of 238 MB/s while KadoP [1] scaled up to 1 GB of data over 50 peers with a throughput of 0.33 MB/s and psiX [9] used 262 MBs of data and 11 computers.

4.2 Query Engine Evaluation

In this Section, we investigate the query processing performance as the data size increases. We use 20 publisher peers, two of which are also consumers, while another publisher is a query peer. The query peer and the two consumers are located in three different French cities. The number of published documents varies from 20 to 500; all documents contribute to the views.

The document used in this experiment is the same as in the previous experiments with a slight difference: its root element *catalog* has only one child, named *camera*. The views defined in the network are the following:

- v_1 is $//catalog_{ID}//camera_{ID}//description_{ID,cont}$
- v_2 is $//catalog_{ID}//camera_{ID}//\{description_{ID}, price_{ID,val}, specs_{ID,cont}\}$

Each document contributes a tuple to each view. The tuples of v_1 are large in size, since the *description* element is the largest element in our documents. A v_2 tuple is quite smaller since it does not store the full camera descriptions. We use two queries: q_1 asks for the $description_{cont}$, $specs_{cont}$ and $price_{val}$ of each *camera*. To evaluate q_1, ViP2P joins the views v_1 and v_2. Observe that q_1 returns full XML elements, and in particular, product descriptions, which are voluminous in our data set. Therefore, q_1 returns roughly all the published data (from 10MB in 20 tuples, to 250MB in 500 tuples). q_2 requires the $description_{ID}$, $specs_{ID}$ and $price_{ID}$ of each *camera*. It is very similar to q_1 with but it can be answered based on v_2 *only*. The returned data is much smaller since there are only IDs and no XML elements: from 2KB in 20 tuples, to 40KB in 500 tuples.

Figure 3 shows the query response time and the time to get the first result for the two queries. The low selectivity query q_1 in Figure 3 (left) takes longer than q_2 (right), due to the larger data transfers and the necessary view join. The time to first result is always constant for both q_1 and q_2 and does not depend on the result size. For q_1, a hash join is used to combine v_1 and v_2, and thus no tuple is output before the view v_2 has been built into the buckets of the hash join. This is done around one second in the case of q_1 and about 300 ms for q_2.

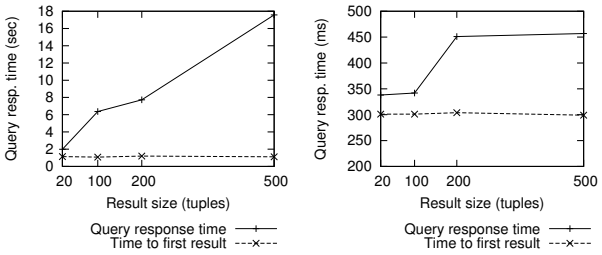

Fig. 3. Query execution time vs. number of result tuples for q_1 (left) and q_2 (right).

The ViP2P query processing engine scales quite linearly answering queries in a wide-area network. The fact that ViP2P rewrites queries into logical plans which are then passed to an optimizer, enables it to take advantage of known optimization techniques used in XML and/or distributed databases.

5 Conclusion and Perspectives

We have presented the ViP2P platform for building and maintaining structured materialized views, and processing queries using the views in a DHT network. Our experiments show that ViP2P outperforms similar systems by several orders of magnitude, in particular for the data publication throughput and the overall volume of data published. Many more experiments are described in our technical report [4]. We currently investigate a distributed version of our automatic view selection algorithm [5]. We also consider multiple-level subscriptions, where some views could be filled with data based on lower-level views.

Acknowledgements We experimented on Grid'5000 (https://www.grid5000.fr). We thank A. Tilea, J. Camacho-Rodríguez, A. Roatis, V. Mishra and J. Leblay for their help. This work was partially supported by ANR 08-DEFIS-004.

References

1. S. Abiteboul, I. Manolescu, N. Polyzotis, N. Preda, and C. Sun. XML processing in DHT networks. In *ICDE*, 2008.
2. A. Bonifati and A. Cuzzocrea. Storing and retrieving XPath fragments in structured P2P networks. *Data Knowl. Eng.*, 59(2), 2006.
3. L. Galanis, Y. Wang, S. R. Jeffery, and D. J. DeWitt. Locating Data Sources in Large Distributed Systems. In *VLDB*, 2003.
4. K. Karanasos, A. Katsifodimos, I. Manolescu, and S. Zoupanos. The ViP2P Platform: Views in P2P. Inria Research Report N^{o}7812, Nov. 2011.
5. A. Katsifodimos, I. Manolescu, and V. Vassalos. Materialized View Selection for XQuery Workloads. In *SIGMOD*, 2012. To appear.
6. K. Lillis and E. Pitoura. Cooperative XPath caching. In *SIGMOD*, 2008.
7. I. Manolescu, K. Karanasos, V. Vassalos, and S. Zoupanos. Efficient XQuery rewriting using multiple views. In *ICDE*, 2011.
8. I. Miliaraki, Z. Kaoudi, and M. Koubarakis. XML Data Dissemination Using Automata on Top of Structured Overlay Networks. In *WWW*, 2008.
9. P. R. Rao and B. Moon. Locating XML documents in a peer-to-peer network using distributed hash tables. *IEEE TKDE*, 21, 2009.
10. A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *ICDSP*, Nov. 2001.
11. G. Skobeltsyn, M. Hauswirth, and K. Aberer. Efficient processing of XPath queries with structured overlay networks. In *OTM Conferences (2)*, 2005.