PinS: Peer to Peer Interrogation and Indexing System

Maria-Del-Pilar Villamil *, Claudia Roncancio, Cyril Labbé
LSR-IMAG Laboratory BP 72, 38402 St. Martin d'Hères, France
e-mail: Firstname.Lastname@imag.fr

Abstract

Distributed Hash Table (DHT) P2P systems provide flexible management of large scale distributed systems. They also provide efficient object localization given its key. However, they don't provide high level query languages to formulate such location queries. Recent efforts have been made to improve such querying capabilities. Nevertheless, most of the proposed solutions are based on special hashing functions or on an additional level of peers. This paper presents PinS, a P2P interrogation and indexation middleware for DHT based P2P systems. It improves data sharing in such systems by supporting declarative queries and some facilities on data management without using particular hash functions or other requirements. Location queries may be conjunctions or disjunctions of conditions including comparison terms, and users may specify some evaluation constraints. Comprehensive or partial answers are provided. PinS supports pre-defined and customized attributes to describe objects and allows sharing objects with restricted access. Additionally, PinS enables several query execution strategies and proposes the use of indexes to improve comparison queries support. Our performance analysis shows the scalability of the system. In particular, the complexity of the evaluation of conjunctive queries is independent of the number of objects registered in the system.

1. Introduction

P2P systems have got a lot of attention in the last years because of their flexibility in managing large scale distributed systems – participants may join and leave easily such a system. Applications like Gnutella [8] made first generation of P2P systems very popular. This systems allow data sharing over a dynamic unstructured set of participants. Object location requests submitted by users are propagated in the system using a flooding strategy. Despite of constraints used to restrict query flooding (e.g., TTL, timeout), this approach uses network resources in a rather greedy way and may lead to unbounded answer times. Proposals as [18, 10] reduce this problem. Concerning queries themselves, we note that such systems do not guarantee comprehensive answers [6], i.e. answers containing all relevant data available in the system.

Structured P2P systems (e.g. CAN [13], Chord [16] and Pastry [14]) attempt to improve query propagation, the comprehensive answers problem and query support in general. Their main purpose remains the management of dynamic and heterogeneous sets of participants. Data sharing – or other kind of applications – may be developed on such P2P systems. Structured P2P systems usually propose a logical organization of peers supported by a Distributed Hash Table (DHT) [9]. Such a table allows to insure scalability and dynamicity of P2P systems and to improve query management (the number of contacted peers is reduced and comprehensive answers may be provided). Our work capitalizes on these results and exploits the advantages of routing mechanisms provided by DHT based systems.

This paper presents a middleware, named PinS, to improve data access in DHT based P2P systems. No hypothesis is made on the type of data (images, video, documents, software components, etc) and we consider attributes (e.g. *Author = "Bach"*) associated to objects as meta-data. PinS proposes a range of functions including, declarative queries (attribute based) and customized data indexing - personalized meta-data on public shared data on top of DHT systems. PinS adapts to P2P systems, several results on data and query management obtained in the database context (e.g. the use of indexes). Unlike some current proposals [1, 2], PinS respects a true P2P architecture, does not rely on centralized catalogs nor on super peers [19] - or other distinguished peers - and does not require hashing functions with particular properties.

PinS proposes several levels of services conditioned by the integration degree with the underlying DHT system. This variety of services composes a spectrum of services having, at one end, a light-weight PinS service offering basic attribute based queries for a black-box P2P system and, at the other end, a powerful service supporting rich querying facilities and customized data and meta-data management.

* Supported by Universidad de Los Andes, Colombie

For this level of service our choices are so that the number of objects registered in the system does not affect the complexity of the evaluation of conjunctive queries. This property is highly appreciable in large scale systems.

This paper is organized as follows. Section 2 reviews main features of DHT systems, proposes a layered point of view of functions they provide and reviews related works on querying in DHT systems. Section 3 overviews the PinS middleware. The main functions are presented in the following sections. Section 4 presents the basic attribute querying facility. Section 5 introduces more advanced query support including comparative queries. Section 6 introduces private objects, customized attributes support and an indexing approach to support comparative queries. Section 7 presents a performances analysis of PinS and Section 8 concludes the paper.

2. Overview on P2P data sharing systems

A hash table is a structure used as index to improve data access. It contains a block collection named bucket array. A hashing function is used to associate a position (the bucket identifier) to a key. Key searches are therefore very efficient. A P2P system can be compared to a bucket array where a peer is a storage unit (bucket). This simile is used by structured P2P to enhance query support. Structured P2P systems are therefore known as Distributed Hash Table Systems (DHTs). As a matter of fact DHT systems are not yet standardized but there are efforts on this direction [5, 3]. Section 2.1 points out the main functionalities offered by DHTs and by storage systems build on them. A two layers view of DHT systems is also introduced. Depending on the level of service, PinS uses some or all functions provided by these layers. Section 2.2 discusses related work on query processing in DHT systems.

2.1. Distributed Hash Table systems

A DHT system splits a key space into zones and assigns each zone to a peer. A key is a point in this space and the object corresponding to this key is stored at the peer whose zone contains this point. Locating an object is reduced to routing to the peers hosting the object. One of the main differences between DHT solutions is their routing geometry. For instance, CAN [13] routes along a d-dimensional Cartesian space, Chord [16] and Pastry [14] along a ring and Viceroy [11] uses a butterfly network. To find an object in such systems, we have to know its key. This key is then easily located among the peers in the system. Such queries are called **location queries**. Its answer contains the peer identifiers where the object is stored. The requester has to get then the object at one of these peers by himself.

Techniques used in the routing process can be seen as a first layer. We named it **Distributed Lookup Service (DLS)** (see figure 1). DLSs provide efficient mechanisms to find peers (named by keys) on a distributed system by managing the routing information (neighbors set of a peer). Their main functions are:

lookup(key): returns the physical identifier of the peer in charge of a key (e.g. on Internet, returns the IP Address).

join(key): registers a new peer in the distributed system and updates routing information.

leave(key): handles the departure of a peer and updates routing information.

neighbors(key): return a list of keys identifying the peers in charge of the neighbor zones.

This basic DLS layer is used to build systems that give semantic to keys and insures data storage with some properties. This can be seen as a second layer, we name **Distributed Storage Service (DSS)**. It is responsible for stored object administration – insertion, migration, etc. Objects migrate from a peer that leaves the systems to one of its neighbors in order to insure object's durability. In fact, object migration and replication (in neighbors of their storage peers) will assure that a stored value never disappears from the DSS. Such systems also offer load balancing and caching strategies. Examples of systems implementing the DSS layer are Past [15] , CFS [4] and DHash [7]. The main external functions of this layer are:

get(key): returns the object (or collection) identified by *key*. It uses *lookup* to locate the peers storing such objects.

put(key, Object): inserts in the system an object identified by its key. It uses *lookup* to identify the peer where the object has to be stored.

It is worth remarking that no function to suppress a key is provided. An object stored in a DSS becomes a shared object. In practice, it is not easy to remove objects as temporally absent peers may have a copy of a deleted object. Coherency problems may arise when such peers are restored.

From the PinS point of view, the main functionalities of DHT and systems using DHT are classified in the Distributed Lookup Service (DLS) and Distributed Storage Service (DSS) layers as shown in Figure 1.

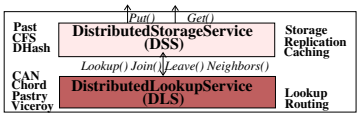

Figure 1. P2P systems basic layers

We should also remark that DHT systems allow to find (or locate) an object given its key and can be extended to

offer real querying facilities. An overview of works on this topic is given in the next section.

2.2. Related works: Querying in P2P systems

DHT systems do not supply, in their basic configuration, keyword search functions or multi attributes and comparative queries. They only offer access to objects giving their key [1]. More powerful queries are difficult to handle since the objects distribution criteria among peers is based on the key. Therefore the routing process cannot be used to solve semantic queries. Several attempts to improve querying capabilities has been performed recently. Most of them concern location queries. Main approaches introduce meta-data to add semantics to stored data. This can be unstructured keywords [7] or a more structured schema like lists of tuples <attribute name, value> [7, 17]. In most cases [2, 7, 17], meta-data are just stored like other data. A key is associated to a meta-data, and a tuple <meta-data, object key> is stored on a peer who is in charge of the meta-data's key. In this way, meta-data of an object is fragmented all over the system.

Meta-data is used in the routing process. [7] supports keywords conjunctions (i.e. *Attribute1= v1 and Attribute2 =v2*) but they don't provide mechanisms for query extensions like keyword disjunctions or the use of comparison operators (e.g. *Attribute1*$\leq$ *v1*). MAAN [2] suggests an alternative to support comparative queries. They use an order preserving hash function [2] to guarantee an organized query propagation and comprehensive answers. To avoid load balancing problems as a consequence of a non uniform distribution a process to generate uniform order preserving hashing functions has been used in MAAN. Nevertheless, all peers that may have requested data are contacted even if they don't have relevant data at that moment. If the number of peers is large this can be prohibitive.

In [17] group by, order by, comparison and join operators are proposed using an additional layer named Range Guards(RG), similar to a super peer layer [19]. Although new operators are proposed, the analysis is specific to each operator and there is no global proposal with all phases involved in the query process and its administration. Additionally, those peers are dedicated to store information about comparisons but there is no information about the process to maintain meta-data up-to-date. This work has constraints such as the knowledge of the lower and higher domain value for attributes involved in comparative queries. The problem with this constraint is that this value may change according to objects stored in the system. Furthermore, it is not clear how several hashing functions proposed for each attribute coexist in the routing process.

Our proposal, PinS, offers three levels of service. The lower level: *DependentPins* is comparable to [7] both proposals support conjunction of (attribute=value) terms and optimize queries by using terms combinations. PinS provides also disjunction of terms, [7] doesn't. [7] works on DHash/Chord systems whereas PinS is a general proposal to work on any DSS and DLS. The second level of service: *IntermediatePinS* supports comparison terms (e.g. $\leq$) as [2, 17] do, but also introduces the like operator (e.g. *Attr like "*Ab*"*) not proposed by them. To support comparison terms, [2] uses an order-preserving hash function whereas PinS does not require a function with particular characteristics. PinS uses the function adopted by the underlying DSS. In [2], the evaluation of queries including a comparative term is directed by this term: the query is successively propagated to all the peers that may potentially have data satisfying such term. Unlike PinS, [2] replicates meta-data on all peers. [17] is oriented to comparison terms and introduces a particular layer of peers to support them. The third level of services: *IndependentPinS* offers a location independent solution using indexes without requiring additional peer characteristics. Higher levels of services in PinS additionally propose partial answers, bounded execution time, access to public and private objects and dynamic attributes. To the best of our knowledge these features are not proposed by related current proposals.

3. Overview of the PinS Middleware

PinS is not bounded to a particular underlying P2P system. It can be used along with any DHT offering functions of the DSS and DLS layers introduced in Section 2. Figure 2 shows a global picture of the system architecture.

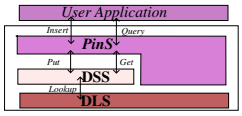

Figure 2. System global architecture with PinS middleware

Shared data may be any type of object that can be identified and described by its meta-data. Data type does not affect PinS behavior as it works on meta-data associated to such objects. Such meta-data is simply composed by <attribute,value> tuples. Let us consider some kind of P2P library for our running example. For a shared book, we may give its title (e.g. *Title="One"*) and publication year

1 Obtained through the hash function

2 A hashing function f preserves an order if for any objects T1 and T2 where $T1 > T2$ then $f(T1) > f(T2)$

(e.g. *Year=1988*). Objects and their meta-data are inserted through a PinS interface who takes in charge data and meta-data registration by delegating some tasks to the underlying DSS. The registration process implies the creation of specific **location data** based on the given <attribute,value> information [3]. Such meta-data contains information about object location and is distributed on several peers. PinS uses the hash function provided by the DSS to place location data and to generate object identifiers. Object identifiers guarantee object content privacy.

PinS querying facility allows users to formulate declarative queries using terms of the form attribute operator value (e.g. *Title="One"*). Attributes are those declared at object registration time or, as we will see later, attributes introduced by the dynamic indexing facility proposed by PinS. The operator used in a term can be $=, \leq, \geq$ and like. Terms using these operators are called **EqTerm**, **IneqTerm** and **LikeTerm** respectively.

A **location query** is a conjunction or a disjunction of terms (e.g. *Year=1988 and Title like "*On*"*). Answers give object placement information; users can then decide which peers to contact to retrieve the objects.

PinS proposes three levels of services (see Figure 3), *Dependent*, *Intermediate* and *Independent*. All of them offer data and meta-data registration but differ in the querying and indexing capabilities they provide and in the tasks delegated to the underlying DSS and DLS layers. Data and meta-data management are decisive tasks. The more PinS participates to these tasks the better querying and indexing facilities are provided.

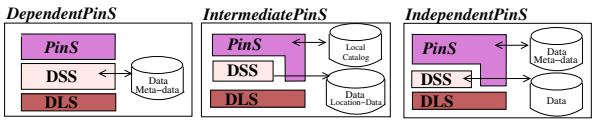

Figure 3. PinS level of services

The DependentPinS service can be considered as a light-weight instance of PinS. It supports a subset of the query language: conjunctions and disjunctions of EqTerms. No other operators are supported. The system provides comprehensive answers. Queries and data and meta-data storage use *put* and *get* DSS functions.

The IntermediatePinS service extends the previous one by introducing IneqTerms and LikeTerms (e.g. *Title = "One" and Year > 1987*). Some query evaluation constraints are now supported: users may specify the answer size and may bound the desired query execution time. Comprehensive or partial answers are provided and several query execution strategies are supported. To do so, **Local catalogs** containing copies of fragments of location data are introduced. PinS, DSS and DLS do meta-data management jointly (i.e. *put*, *get* and *lookup* functions).

The IndependentPinS service improves IntermediatePinS in several ways. It is the more powerful level of service.

- PinS complete query language is supported in an efficient way: comparison queries are optimized and supported in a general way (e.g. *Year > 1987*) by using a B-tree [12] in the localization process.
- Object location can be personalized in order to provide access constraints. Private and public objects may be managed. Queries access both categories of data in a uniform way.
- Meta-data – attributes – can be personalized. Customized attributes can be introduced by users as a complement to common predefined attributes. Such attributes are considered as dynamic and may index public and private shared objects.

The following sections present PinS services in an incremental way.

4. Initial querying facilities

PinS initial service, DependentPinS, provides location queries of the form

$$attribut1 = v1\ [and/or\ atribut2 = v2]*$$

For example, *Title = "One hundred years of solitude" or Author = "Gabriel Garcia Marquez"*. To support this querying facility, DependentPinS performs data and meta-data registration but relies on the underlying DSS for their storage (no direct access to the DLS is made by PinS). Data and meta-data are distributed in the P2P system. DependentPinS may be easily used on top of existing DSS to provide a form of declarative location queries.

4.1. Object registration

Registering an object O1, also includes its meta-data registration. Figure 4 illustrates the registration of the book "One hundred years of solitude". Such registration uses the hash function provided by the DSS. **DSSHashF** denotes this function in the following. A key returned by this function will be noted with the prefix **Id** – e.g. IdObj1. To make short, using our example of Figure 4, we will simply write the title as OYS and the author as GGM.

PinS's object registration is as follows:

3 We suppose attribute names are decided/accepted by the community using the system. Proposals for handling schema heterogeneity could be used but are out of the scope of this paper

1. Call DSSHashF(O1) to obtain the object identifier – IdObj1. In figure 4 the object identifier is 123.

2. For each term Ti composing O1's meta-data, call DSSHashF (Ti) to generate its identifier - let's say IdTi. Figure 4 shows this step for three terms. For example 510 is the identifier of term *Date=1967*.

3. Register O1 in the DSS with access key IdObj1 - *put (IdObj1,O1)*. See point 3 in figure 5.

4. Register location data for object O1 using its meta-data: for each term Ti, register IdObj1 in the DSS with access key IdTi - *put(IdTi, IdObj1)*. IdObj1 is the associated value, not the key. See point 3 in figure 5. A peer storing location data plays the role of **location peer**.

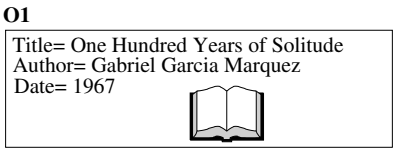

Figure 4. Example of object description

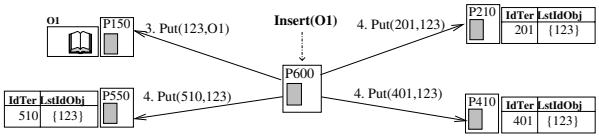

Figure 5. Object registration

4.2. Location queries evaluation

Any peer may receive location queries. The contacted peer, referred as **access peer** in the following, handles the query evaluation process as follows :

1. For each term Ti composing the query, call DSSHashF(Ti) to obtain its identifier IdTi. This is similar as point 2 in figure 4 where 510 is a term identifier.

2. Obtain from the DSS all location data associated to each term identifier - call *get(IdTi)*. In Figure 6, *Get(510)* is an example of this step.

3. Extract the set of object identifiers from the obtained location data. Calculate query's terms conjunctions with intersections and terms disjunctions with unions - call CalculateAnswer function. See point 3 in Figure 6.

4. Compose and return the complete query answer as a set of object identifiers. See point 4 in Figure 6.

The set of returned object identifiers is a comprehensive answer. Users can then retrieve objects themselves by using the identifiers (call to *get*).

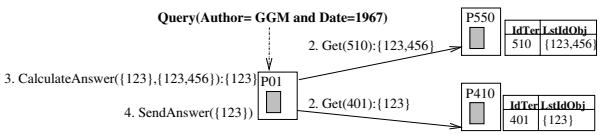

Figure 6. Location query process

Efficiency depends on the characteristics of the DSS and on the capabilities of the access peer. The DSS takes in charge peers communication and data availability. However, two problems related to the way Pins uses the DSS may appear. The first one is related to terms frequently used in location queries as they may overload the location peer. Replication and load balancing strategies provided by the DSS reduce this problem but are not a definitive solution. The second problem is related to the size of the answer: either a term retrieves numerous identifiers or the final size is too big. To deal with this issue, DependentPinS provides an optimization based on meta-data registration. The object registration process will also register conjunctions of terms composing the meta-data of the object. This reduces the number of messages and may reduce the number of object identifiers returned by a term without affecting the final answer. This optimization does not improves disjunctive queries.

5. Comparative queries and partial answers

PinS intermediate service, IntermediatePins, improves querying facilities by introducing comparative terms in queries (see Section 5.1) and providing comprehensive and partial answers. Query execution strategies (see Section 5.2) are introduced to distribute load over different peers.

5.1. Enhancing querying capabilities

Querying Capabilities: Queries based on EqTerms are well suited for users knowing the exact values for the attributes used in their queries. However, if this is not true

comparison terms ($\leq, \geq, like$) are very helpful. DependentPins uses the DSSHashF to generate terms identifiers. Such hashing functions usually do not preserve a data order. Moreover, it is not possible to identify the attribute and the value hidden behind a term identifier. These two issues prevent DependentPins from supporting $IneqTerm$ evaluations.

IntermediatePins supports queries as in DependentPins and queries of the form:

$$EqTerm\ and\ (IneqTerm|LikeTerm)\ [and/orTerm]*$$

where Term denotes either an EqTerm, an IneqTerm or a LikeTerm.

In some cases, users do not need comprehensive answers and a partial answer including a specified number of objects may be enough. Problems providing comprehensive or partial answers have been studied and solved according to P2P system characteristics. First generation P2P systems had problems to guarantee comprehensive answers because of their constraints to restrict query flooding (e.g.TTL). On the other hand, structured P2P systems guarantee comprehensive answers. As in DependentPins, IntermediatePins takes advantage of DHTs to provide comprehensive answers. Partial answers support is still to be done and is introduced in IntermediatePins. Query answers are composed by partial answers returned by different peers. The problem is to be able to indicate the number of answers a peer has to send to guarantee load balancing and take into account the data they handle. IntermediatePins supports query evaluation constraints allowing users to indicate the approximate answer size they desire (number of items) and to bound maximal execution time (e.g. timeout).

Local catalogs: IntermediatePinS, as DependentPins, handles location data fragmented and distributed in the P2P system to offer a scalable solution. As this meta-data is not enough to support the new querying capabilities, IntermediatePins introduces local catalogs in peers to maintain a relation between terms and references to locally stored objects. For example, let's consider peer P150 storing the book entitled OYS (Figure 5). The local catalog of P150, has a record for each term describing the book. Such a record contains the term identifier, the attribute name, the attribute value and the list of local objects satisfying the term, see figure 7. Meta-data includes local catalogs, entirely handled by the Pins layer, and location data, handled by the DSS layer.

Local Catalogs management cannot be delegated to the DSS because we need to ensure that such catalogs are located together with the referenced objects in the same peer. IntermediatePinS uses event notification (events related to object migration/duplication) provided by the DSS to guarantee coherency.

5.2. Query execution strategy

IntermediatePins introduces query execution strategies to support the new querying functionalities. Such strategies are supported calling the DSS *get* function and the DLS *lookup* function. Peer roles are distinguished, **location peers** store location data whereas **storage peers** store data and the corresponding local catalog. Any peer can play several roles: access, location and/or storage peer. A query execution strategy includes propagation, evaluation and answer retrieval phases. Each phase may be performed in different ways as discussed in the following.

Propagation Phase: When a query arrives, IntermediatePins chooses a **query coordinator**. The access peer can be coordinator or can delegate the coordination to one of the location peers. For a given query, the location peers are those storing the location data corresponding to the terms used in the query. The coordinator contacts the (other) location peers, delegates to each one a part of the query and calculates final answers. It uses the initial query and its execution constraints (i.e. timeout/number of answers) to decompose the query. A contacted location peer may behave as **sub-coordinator** and contact other location peers to resolve its part of the query.

Choices in this phase concern coordinator selection and query decomposition. Metrics to choose the coordinator can be the nearest peer to the access peer or the peer in the middle point between all the location peers. Query decomposition allows partial answers and parallel evaluations. Query decomposition can be based on the type of terms (e.g. $\leq, \geq$, like, =) or on the number of terms assigned to a peer. Execution constraints are considered in the decomposition and propagated as necessary. To handle the number of desired answers the decomposition strategy may be independent of the contacted peer (e.g. all peers are asked to retrieve the same number of answers) or dependent of past peer behavior (e.g. answers returned in the past). Concerning specified timeouts (if any), the initial query timeout is reduced a delta of time before the propagation to other peers.

Evaluation Phase: A location peer contacted to evaluate a query can do it locally or in cooperation with storage peers. This phase is composed of three parts. They are illustrated in Figure 7.

1. Use the local catalog to include relevant local objects in the answer, call SearchLocalCatalog function. A partial answer may be returned.

2. Use location data to resolve the (part of) query it is in charge for, call SearchLocationData function.

3. Delegate the query evaluation to the storage peers, call Evaluate function.
Part 3 is mandatory if the query includes Ineq or Like Terms. These terms are the last terms to be evaluated.

The Answer Retrieval Phase decides the path to be used to return the answer if it is not in the access peer. Choices depend on the preceding phases. Peers participating to the evaluation phase can return the answer directly to the access peer or along the path used to send the query. The second option allows to hide the access peer identifier to other participant peers and enables eventual answer caching in the contacted peers.

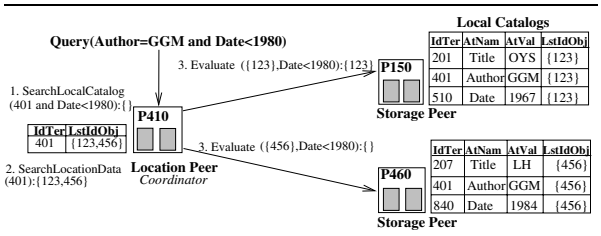

Figure 7. Query evaluation phase

Some alternatives for the phases involved in a query strategy were presented. The choice of an alternative should be based on metrics such as the query characteristics, access and location peers capacity and historic peer behavior (e.g. response time).

6. Dynamic indexation and improves in comparative queries

This section introduces new features proposed in IndependentPins. It improves preceding proposals by including private objects and dynamic attributes allowing customized indexing and querying. IndependentPins also offers general and efficient support of comparative queries.

6.1. Private and public objects

Current proposals allow data sharing in a large scale way. Shared objects are "public" and any user may access to them. DependentPins and IntermediatePins handle public objects exclusively. IndependentPins extends the system to include private objects. Meta-data of such objects is registered by IndependentPins but the objects themselves are stored by the owner peer. They are not stored by the underlying DSS and don't benefit of DSS's management (e.g. migration, duplication policies) as public objects do. The global space of shared objects is so composed by public objects and several sets of private objects. Location queries behave uniformly on this space and answers include identifiers of public and private objects. Effective access to private objects have to be requested to their owner who can deliver objects in a controlled way. Private objects can be suppressed by its owner. Public objects are not supposed to be suppressed. Applications sharing objects with public/private status are varied. For example, consider that shared objects are software components. On one hand, software freeware is shared by a community where distribution politics are well specified and respected. On the other hand, for commercial software, providers share freely information about its existence and characteristics to everybody, but only buyers can access the software components themselves.

IndependentPins's meta-data is extended to include object status – private or not – and the storage peer address (IPAddress) for private objects. Unlike IntermediatePins, meta-data is handled by IndependentPins (not by the DSS). Coherency between private objects and their meta-data is insured by Pins. Two strategies, pull and push based, are proposed: in the push based strategy, the storage peer initiates meta-data update after a local object deletion. It updates the local catalog, searches the related location peers (using *lookup()* of the DLS) and notifies them the delete event. Location peers will then remove related meta-data. In the pull based strategy, location peers ask periodically for delete events and update location data asynchronously.

6.2. Improving queries with comparative terms

IntermediatePins supports queries including Ineq/Like Terms but it requires queries to have at least one EqTerm to perform efficiently. IndependentPins improves the support of Ineq/Like Terms by using an additional index based on B-trees [12]. Ineq/Like Terms are there supported in a general and efficient way.

B-tree indexes are handled with search, insert and delete B-tree standard algorithms. An index on an attribute A is created on the peer related to attribute A according to the DSS hash function and placement strategy - peer returned by lookup(DSSHashF(A)). Such a peer is called **attribute storage peer** from now on. Indexes store information independent of objects location. This allows a search strategy where only peers with objects satisfying the requested Ineq/Like Term are contacted.

A B-tree node is composed by attribute's values (e.g. 1969) and term identifiers containing these values (e.g. DSSHashF(Date=1969)), see figure 8. Terms identifiers represent objects in the system satisfying the term (e.g. Date=1969).

Term identifiers allow IndependentPins to contact peers concerned by the Ineq/Like Term of the query and guarantee object location independence. Object's migration do not invalidate indexes. Indexes are updated when objects are inserted/deleted. The location peer in charge of the related term (e.g. Date=1969) sends the update event to the attribute storage peer who updates the index.

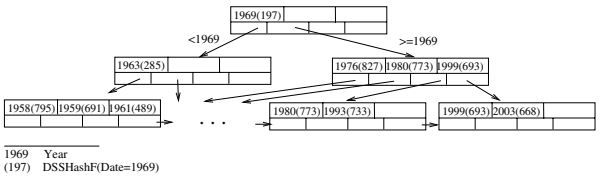

Figure 8. B-tree on date attribute

The query evaluation process is changed to include indexes. Let's consider the query $Date > 1980$ to illustrate this process:

1. Find the **attribute storage peer** for Date - Call lookup(DSSHashF($Date$)). Let's say PeerK.

2. On PeerK, use the B-tree search algorithm to obtain the terms identifiers satisfying $Date > 1980$. Using example of figure 8, the peers storing relevant location data are 733, 693 and 668.

3. Contact location peers for such terms and continue the evaluation process as described in section 5.

This evaluation process is used for IneqTerms and LikeTerms.

6.3. Dynamic attributes

Dynamic attributes are proposed to allow users to declare their own attributes on shared objects (public or private ones). In this way objects are described by the common defined attributes and authorized users may introduce temporary customized attributes. For example, if a user desires to indicate the bestseller book of the month, (s)he can associate the term Bestseller="May 2004" to the selected book. Such attributes are considered as dynamic because they have a limited lifetime specified by the user when the attribute is declared. Queries may involve predefined attributes (as before) and current dynamic attributes. It is important to point out that no modification is required on the shared objects and even storage peers do not need to be aware of dynamic attributes. IndependentPins considers dynamic attributes as location data but does not include them in local catalogs. During query evaluation, terms on dynamic attributes are evaluated in the corresponding location peers. Location data for dynamic attributes has as key the term identifier (i.e. DSSHashF(Bestseller="May 2004")). The associated value contains the identifier of the concerned object and the attribute's expiration time.

Dynamic attributes are deleted by location peers according to their expiration time. Eager and lazy approaches are proposed. The first one deletes expired attributes daily. The lazy approach simply does not include expired attributes when the peer is contacted for a query. An event (e.g. query, insertion or temporal event) is chosen to trigger expired attributes deletion. Coherency is guaranteed by both approaches.

7. Performance analysis

PinS uses the *get* and *lookup* functions of underlying DSS/DLS layers to provide querying facilities. Therefore PinS performances depend on the performances of such functions and on the query evaluation strategy PinS implement. This section presents a performance analysis of PinS services in terms of the number of *get/lookup* calls made to evaluate a query. The complexity of both functions is usually the same in a DHT system. For example for Pastry or Chord it is on $O(logN)$, where N is the number of peers in the system. So, without lost of generality, in the following, we do not distinguish one function from the other and calculate globally the number of *get* and *lookup* calls. We note it as *get* calls.

Definitions Let N_{ob} be the number of objects registered in the system.
Let A_i be an attribute with domain $DomA_i$ and let $|A_i|$ be the cardinality of $DomA_i$.
An IneqTerm $Tineq_i$ is of the form $A_i < v_i$ or $A_i > v_i$. Let $|Tineq_i|$ be the cardinality of $Tineq_i$: the number of distinct values (in $DomA_i$) satisfying the condition.

Let us now analyze the performance of PinS considering conjunctive queries using EqTerms and/or IneqTerms and comprehensive answers.

Conjunction of EqTerms Let Teq_i be an EqTerm ($A_i = v$ where $v \in DomA_i$) and consider a query composed of k EqTerms $Q_1 = Teq_1 and\ \ Teq_2$ ... $and\ Teq_k$. For such queries, the query evaluation strategy analyzed here is the same in DependentPins, IntermediatePins and IndependentPins. In the basic approach (no optimization) Pins performs one *get* call per term appearing in the query. For Q_1 the maximum number of *gets* is k. If predefined conjunction of terms is stored as meta-data, the number of *gets* may fall to 1. This would be the best possible case in DependentPins and IntermediatePins. It is interesting to remark that in any case the number of *get* calls used to evaluate the query is independent of N and of N_{ob}.

Conjunction of EqTerms and IneqTerms Let us consider a query composed of k EqTerms and l IneqTerms: $Q_2 = Teq_1\ \ and\ \ ...\ Teq_k\ \ and\ \ Tineq_1\ \ and\ \ Tineq_l$. Such a query can be evaluated by IntermediatePins and IndependentPins, but not by DependentPins. The query evaluation strategy includes a phase for the conjunction of EqTerms and a phase for the conjunction of IneqTerms.

Let $SF(Teq_i)$ be the selectivity factor of Teq_i - the probability of an object to satisfy the condition. Supposing an homogeneous distribution of values $SF(Teq_i) =$

$\frac{1}{|A_i|}$, the average number of objects satisfying the condition stated by Teq_i is

$$N_{ob} * SF(Teq_i) = N_{ob} * \frac{1}{|A_i|}$$

The average number of objects satisfying $Teq_1 \; and \; ... \; Teq_k$ is $N_{ob} * \prod_{i=1}^{k} \frac{1}{|A_i|}$.

Now we can estimate the number of *get* calls required to evaluate Q_2.

For IntermediatePins it is $k + N_{ob} * \prod_{i=1}^{k} \frac{1}{|A_i|}$. The first part, k, corresponds to the evaluation (worst case) of the EqTerms part of the query. For each answer retrieved by this part, Pins performs a *get* call. This is represented by the second part of the formula. We note that the evaluation cost[4] for IntermediatePins depends on the number of objects in the system – N_{ob} – and the selectivity factor of the EqTerms part of the query. The most selective is this factor the better are performances. On the other hand, the selectivity factor of the IneqTerms and their cardinality do not affect the performances.

For IndependentPins the maximum number of *get* calls required to evaluate Q_2 is $k + l + \sum_{i=1}^{l} |Tineq_i|$. The first part, k, corresponds to the evaluation (worst case) of the EqTerms part of the query. Then for each IneqTerm, Pins has to locate the B-tree associated to the attribute. This leads to (maximum) l *get* calls, one per IneqTerm. The B-tree exactly indicates the location peers handling data that satisfy each IneqTerm. There is one *get* call per registered distinct value. The maximum number of *get* calls to be performed is $\sum_{i=1}^{l} |Tineq_i|$.

It is interesting to remark that the cost of IndependentPins does not depend on the number of objects in the system. These results confirm the scalability of the proposal.

Figures 9 and 10 show the facts we discussed before. Figure 9 shows the number of *get* calls in regard to N_{ob}. Plain lines represent IntermediatePins performances to evaluate $Q_3 = Teq_1 \; and \; Tineq_1$ with three different $SF(Teq_i)$: 0.001, 0.01 and 0.5. Dotted lines represent IndependentPins performances to evaluate Q_3 with three different $|Tineq_1|$. [5]

Figure 10 shows the number of *get* calls in regards to different types of queries. The x-axis is labelled with two items indicating respectively the number of EqTerms (k) and IneqTerms (l) appearing in the query. For instance, 1-2 means a query with one EqTerm and two IneqTerms. N_{ob} is 100000 and $SF(Teq_i) = 1/50$. Globally we can conclude that for queries using high selective EqTerms and IneqTerms with large cardinality, the strategy proposed by IntermediatePins is the most appropriated. On the other hand, queries with low selective EqTerms (or no EqTerms) are better served by IndependentPins.

4 in number of *get* calls

5 $|Tineq_1|$ does not affect IntermediatePins performances whereas $SF(Teq_i)$ does not affect IndependentPins performances

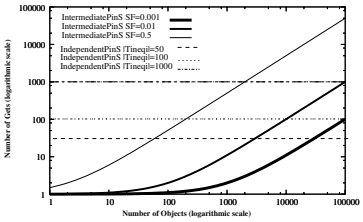

Figure 9. Number of Gets for a query with different selectivity factors according to number of objects

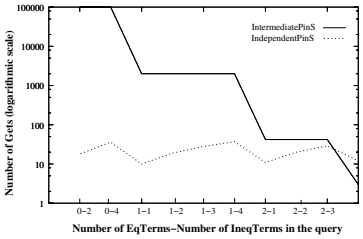

Figure 10. Number of Gets for different types of queries

8. Conclusions and research perspectives

Distributed Hash Table based P2P systems provide appropriate management of large scale loosely coupled peers. They are ideal for key searches but don't provide, in their basic configuration, a high level query language to find (or locate) stored data. Recent proposals improve this feature, but most of the proposed solutions require special hashing functions (e.g. order preserving functions) or rely on distinguished peers. PinS proposes a set of functionalities to improve data sharing on DHT P2P systems without such constraints. Meta-data, sets of attributes and their values, is associated to shared objects. Location queries may be formulated using a language supporting conjunctions and disjunctions of terms (attribute operator value). Supported operators are $=, \leq, \geq$ or $like$. Users may also specify if they need a comprehensive answer or not, and introduce query evaluation constraints. In order to serve different data sharing requirements, PinS distinguishes public and private objects and allows users to declare their own dynamic attributes. Unlike global defined attributes, dynamic

ones have a bounded lifetime and are intended to facilitate customized access without affecting shared objects. From user's point of view location queries perform uniformly on private/public objects and global/dynamic attributes. PinS is destined to DHT P2P systems but is independent of their routing structure. To clarify levels of interaction with the underlying DHT system, we distinguished the Distributed Storage Service (DSS) and the Distributed Lookup Service (DLS). PinS has three levels of service: dependent, intermediate and independent. Each level offers more functionality than the preceding one, and access the DSS and DLS in different ways. PinS relies on the underlying DHT system and adopts database techniques (e.g., fragmentation, indexes) to provide the new functionalities. It enables the use of different query evaluation strategies, it uses B-tree indexes in an original way to support efficiently comparative queries. Our performances analysis is promising. IndependentPins offers a solution where the complexity of the query evaluation is independent of the number of objects in the system. This is an important characteristic for data sharing in large P2P systems. The implementation of our proposal is ongoing-work. Future work is related to this implementation and to a more-in-deep evaluation of the system. Caching strategies and replication related issues should be further investigated. The current proposal entirely relies on the replication strategy provided by the underlying DHT system. We also intend to investigate the cooperation of different levels of PinS services in the same P2P system according to peers characteristics.

Acknowledgments We would like to thank O. Richard, Y. Denneulin and G. Da Costa for fruitful discussions on this work and A.Mislove for his detailed answers to questions about Past and Pastry. This work is part of the NODS (Networked Open Database Services) project [6].

References

[1] I. Brunkhorst, H. Dhraief, A. Kemper, W. Nejdl, and C. Wiesner. Distributed Queries and Query Optimization in Schema-Based P2P Systems. *IPTPS*, 2003.

[2] M. Cai, M. Frank, J. Chen, and P. Szekely. MAAN: A Multi-Attribute Addressable Network for Grid Information Services. *4th Int'l WS on Grid Computing*, 2003.

[3] F. Dabek, E. Brunskill, M. Kaashoek, D. Karger, R. Morris, I. Stoica, and H. Balakrishnan. Building Peer-to-Peer Systems with Chord, a Distributed Lookup Service. In *HotOS-VIII*, 2001.

[4] F. Dabek, M. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. of the 18th ACM SOSP*, 2001.

[5] F. Dabek, B. Zhao, P. Druschel, and I. Stoica. Towards a common API for structured peer-to-peer overlays . *IPTPS*, 2003.

[6] N. Daswani, H. Garcia-Molina, and B. Yang. Open Problems in Data-Sharing Peer-to-Peer Systems. In *Int'l Conf. on Database Theory*, 2003.

[7] O. D. Gnawali. A Keyword-Set Search System for Peer-to-Peer Networks. Master thesis, Massachusetts Institute Of Technology, 2002.

[8] The Gnutella Protocol Specification v0.41. http://www9.limewire.com/developer/gnutella_protocol_0.4.pdf.

[9] M. Harren, J. Hellerstein, R. Huebsch, B. Loo, S. Shenker, and I. Stoica. Complex queries in dht-based peer-to-peer networks. IPTPS, 2002.

[10] Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker. Search and replication in unstructured peer-to-peer networks. In *Proc. 16th Int'l Conf. on Supercomputing*, 2002.

[11] D. Malkhi, M. Naor, and D. Ratajczak. Viceroy: A Scalable and Dynamic Emulation of the Butterfly. In *Proc. 21st ACM Symposium on Principles of Distributed Computing*, 2002.

[12] H. Molina, J. Ullman, and J. Widom. *Database system implementation*. Prentice Hall, 2000.

[13] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content Addressable Network. In *ACM SIGCOMM*, 2001.

[14] A. Rowstron and P. Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. *LNCS-2218*, 2001.

[15] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Symposium on Operating Systems Principles*, 2001.

[16] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A Scalable "Peer-To-Peer" Lookup Service for Internet Applications. In *ACM SIGCOMM*, 2001.

[17] P. Triantafillou and T. Pitoura. Toward a unifying framework for complex query processing over Structured Peer-to-Peer Data Networks. *VLDB WS on Databases, Information Systems, and Peer-to-Peer Computing*, 2003.

[18] B. Yang and H.Garcia-Molina. Efficient search in peer-to-peer networks. Technical report, http://dbpubs.stanford.edu:8090/pub/2001-47, 2001.

[19] B. Yang and H.Garcia-Molina. Designing a Super-peer Network. *IEEE 19th Int'l Conf. Data Engineering*, 2003.

6 http://www-lsr.imag.fr/Les.Groupes/STORM/