A Unified Peer-to-Peer Database Framework for Scalable Service and Resource Discovery

Wolfgang Hoschek

CERN IT Division
European Organization for Nuclear Research
1211 Geneva 23, Switzerland
wolfgang.hoschek@cern.ch

Abstract. In a large distributed system spanning many administrative domains such as a Data Grid, it is desirable to maintain and query dynamic and timely information about active participants such as services, resources and user communities. However, in such a database system, the set of information tuples in the universe is partitioned over many distributed nodes, for reasons including autonomy, scalability, availability, performance and security. It is not obvious how to enable general-purpose discovery query support and collective collaborative functionality that operate on the distributed system as a whole, rather than on a given part of it. Further, it is not obvious how to allow for search results that are fresh, allowing dynamic content. It appears that a Peer-to-Peer (P2P) database network may be well suited to support dynamic distributed database search, for example for service discovery. In this paper, we devise the *Unified Peer-to-Peer Database Framework (UPDF)*, which allows to express specific applications for arbitrary query languages (e.g. XQuery, SQL) and node topologies, and a wide range of data types, query response modes (e.g. Routed, Direct and Referral Response), neighbor selection policies, pipelining characteristics, timeout and other scope options.

1 Introduction

Grid technology attempts to support flexible, secure, coordinated information sharing among dynamic collections of individuals, institutions and resources. This includes data sharing but also includes access to computers, software and devices required by computation and data-rich collaborative problem solving [1]. Grids are cooperative distributed Internet systems characterized by large scale, heterogeneity, lack of central control, multiple autonomous administrative domains, unreliable components and frequent dynamic change. These and other advances of distributed computing are necessary to increasingly make it possible to join loosely coupled people and resources from multiple organizations.

For example, the next generation Large Hadron Collider project at CERN, the European Organization for Nuclear Research, involves thousands of researchers and hundreds of institutions spread around the globe. A massive set of computing resources is necessary to support it's data-intensive physics analysis applications, including thousands of network services, tens of thousands of CPUs, WAN Gigabit networking as well as Petabytes of disk and tape storage [2]. To make collaboration viable, it was decided to share in a global joint effort - the European Data Grid (EDG) [3–6] - the data and locally available resources of all participating laboratories and university departments.

An enabling step towards increased Grid software execution flexibility is the (still immature and hence often hyped) *web services* vision [3, 7] of distributed computing where programs are no longer configured with static information. Rather, the promise is that programs are made more flexible, adaptive and powerful by querying Internet databases (registries) at runtime in order to discover information and network attached third-party building blocks. Services can advertise themselves and related metadata via such databases, enabling the assembly of distributed higher-level components. For example, a data-intensive High Energy Physics analysis application sweeping over Terabytes of

data looks for remote services that that exhibit a suitable combination of characteristics, including appropriate interfaces, operations and network protocols as well as network load, available disk quota, access rights, and perhaps Quality of Service and monetary cost. It is thus of critical importance to develop capabilities for rich service discovery as well as a query language that can support advanced resource brokering.

More generally, in a distributed system, it is often desirable to maintain and query dynamic and timely information about active participants such as services, resources and user communities. As in a data integration system [8–10], the goal is to exploit several independent information sources as if they were a single source. However, in a large distributed database system spanning many administrative domains, the set of information tuples in the universe is partitioned over one or more distributed nodes, for reasons including autonomy, scalability, availability, performance and security. It is not obvious how to enable powerful discovery query support and collective collaborative functionality that operate on the distributed system as a whole, rather than on a given part of it. Further, it is not obvious how to allow for search results that are fresh, allowing time-sensitive dynamic content. Distributed (relational) database systems [11] assume tight and consistent central control and hence are infeasable in Grid environments, which are characterized by heterogeneity, scale, lack of central control, multiple autonomous administrative domains, unreliable components and frequent dynamic change. It appears that a Peer-to-Peer (P2P) database network may be well suited to support dynamic distributed database search, for example for service discovery.

The overall P2P idea is as follows. Rather than have a centralized database, a distributed framework is used where there exist one or more autonomous database nodes, each maintaining its own data. Queries are no longer posed to a central database; instead, they are recursively propagated over the network to some or all database nodes, and results are collected and send back to the client. A node holds a set of tuples in its database. Nodes are interconnected with links in any arbitrary way. A link enables a node to query another node. A *link topology* describes the link structure among nodes. The centralized model has a single node only. For example, in a service discovery system, a link topology can tie together a distributed set of administrative domains, each hosting a registry node holding descriptions of services local to the domain. In other examples, nodes may support replica location [12], replica management and optimization [13, 14], interoperable access to grid-enabled relational databases [15], gene sequencing or multi-lingual translation, actively using the network to discover services such as replica catalogs, remote gene mappers or language dictionaries. Several link topology models covering the spectrum from centralized models to fine-grained fully distributed models can be envisaged, among them single node, star, ring, tree, graph and hybrid models [16]. Figure 1 depicts some example topologies.

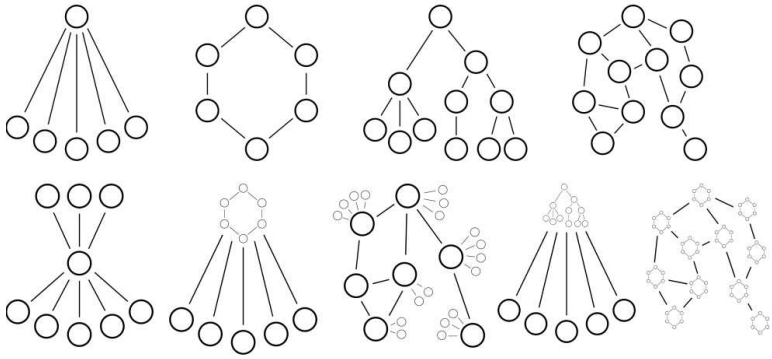

Fig. 1. Example Link Topologies [16].

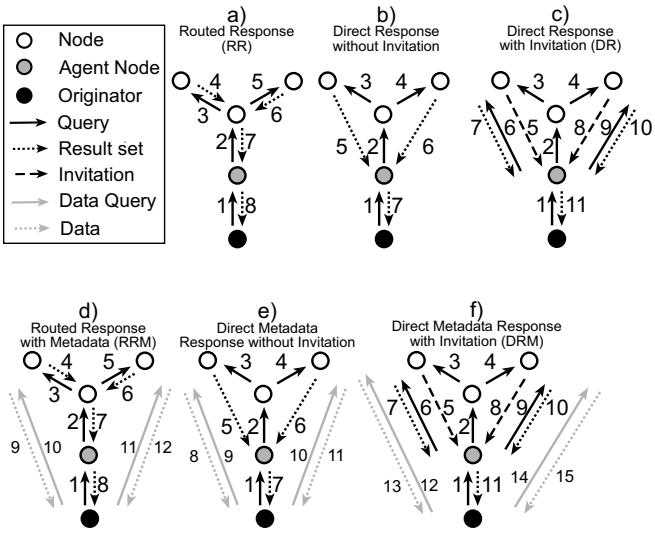

Fig. 2. Peer-to-Peer Response Modes [21].

In any kind of P2P network, nodes may publish themselves to other nodes, thereby forming a topology. In a P2P network for service discovery, a *node* is a service that exposes *at least* interfaces for publication and P2P queries. Here, nodes, services and other content providers may publish (their) service descriptions and/or other metadata to one or more nodes. Publication enables distributed node topology construction (e.g. ring, tree or graph) and at the same time constructs the database to be searched. For example, based on our *Web Service Discovery Architecture (WSDA)* [17], we have introduced a registry node [18] for service discovery that allows to publish and query dynamic tuples, which are annotated multi-purpose soft state data containers that may contain a piece of arbitrary *content* and allow for refresh of that content at any time. Examples of content include a service description expressed in WSDL [19], a Quality of Service description, a file, file replica location, current network load, host information, stock quotes, etc. For a detailed discussion of a wide range of discovery queries, their representation in the XQuery [20] language, as well as detailed motivation and justification, see our prior studies [3]. In other examples, a node may support replica management optimization, gene sequencing or multi-lingual translation, actively using the network to discover services such as replica catalogs, remote gene mappers or language dictionaries.

When any *originator* wishes to search the P2P network with some query, it sends the query to an *agent node*. The node applies the query to its local database and returns matching results; it also forwards the query to select *neighbor nodes*. These neighbors return their local query results; they also forward the query to select neighbors, and so on. In [21] four techniques to return matching query results to an originator are characterized, namely Routed Response, Direct Response, Routed Metadata Response, and Direct Metadata Response (see Figure 2). Under *Routed Response*, results are fanned back into the originator along the paths on which the query flowed outwards. Each (passive) node returns to its (active) client not only its own local results but also all remote results it receives from neighbors. Under *Direct Response*, results are not returned by routing through intermediary nodes. Each (active) node that has local results sends them directly to the (passive) agent, which combines and hands them back to the originator. Interaction consists of two phases under *Routed Metadata Response and Direct Metadata Response*. In the first phase, routed responses or direct responses are used. However, nodes return only small metadata results. In the second phase, the

originator selects which data results are relevant. The originator directly connects to the relevant data sources and asks for data results.

Simple queries for lookup by key are assumed in most P2P systems such as Gnutella [22], Freenet [23], Tapestry [24], Chord [25] and Globe [26], leading to highly specialized *content-addressable* networks centered around the theme of distributed hash table lookup. Simple queries for exact match (i.e. given a flat set of attribute values find all tuples that carry exactly the same attribute values) are assumed in systems such as SDS [27] and Jini [28]. Others such as LDAP [29] and MDS [30] consider simple queries from a hierarchical namespace. None support rich and expressive general-purpose query languages such as XQuery [20] and SQL [31]. The key problems then are:

- *What are the detailed architecture and design options for P2P database searching? How should a P2P query processor be organized? What query types can be answered (efficiently) by a P2P network? What query types have the potential to immediately start piping in (early) results? How can a maximum of results be delivered reliably within the time frame desired by a user, even if a query type does not support pipelining? How can loops be detected reliably using timeouts? How can a query scope be used to exploit topology characteristics in answering a query?*

- *Can we devise a unified P2P database framework for general-purpose query support in large heterogeneous distributed systems spanning many administrative domains? More precisely, can we devise a framework that is unified in the sense that it allows to express specific applications for a wide range of data types, node topologies, query languages, query response modes, neighbor selection policies, pipelining characteristics, timeout and other scope options?*

In this paper, we take the first steps towards unifying the fields of database management systems and P2P systems, which so far have received considerable, but separate, attention. We extend database concepts and practice to cover P2P search. Similarly, we extend P2P concepts and practice to support powerful general-purpose query languages such as XQuery [20] and SQL [31]. As a result, we answer the above questions by proposing the *Unified Peer-to-Peer Database Framework (UPDF)*.

This paper is organized as follows. Section 2 unifies query processing in centralized, distributed and P2P databases. A theory of query processing for queries that are (or are not) *recursively partitionable* is proposed, which directly reflects the basis of the P2P scalability potential. We discusses for which query types the originator has the potential to immediately start piping in results (at moderate performance rate). Other query types must wait for a long time until the first result becomes available (the full result set arrives almost at once, however). Section 3 and 4 discuss loop and abort timeouts, as well as query scoping. Section 5 compares our work with existing research results. Finally, Section 6 concludes this paper.

2 Query Processing

In a distributed database system, there exists a single local database and zero or more neighbors. A classic centralized database system is a special case where there exists a single local database and zero neighbors. From the perspective of query processing, a P2P database system has the same properties as a distributed database system, in a recursive structure. Hence, we propose to organize the P2P query engine like a general distributed query engine [32, 11]. A given query involves a number of operators (e.g. SELECT, UNION, CONCAT, SORT, JOIN, SEND, RECEIVE, SUM, MAX, IDENTITY) that may or may not be exposed at the query language level. For example, the SELECT operator takes a set and returns a new set with tuples satisfying a given predicate. The UNION operator computes the union of two or more sets. The CONCAT operator concatenates the elements of two or more sets into a list of arbitrary order (without eliminating duplicates). The IDENTITY operator returns its input set unchanged. The semantics of an operator can be satisfied by several operator implementations,

using a variety of algorithms, each with distinct resource consumption, latency and performance characteristics. The query optimizer chooses an efficient query execution plan, which is a tree plugged together from operators. In an execution plan, a parent operator consumes results from child operators.

Template Query Execution Plan. *Any* query `Q` within our query model can be answered by an agent with the *template execution plan* `A` depicted in Figure 3. The plan applies a local query `L` against the tuple set of the local database. Each neighbor (if any) is asked to return a result set for (the same) neighbor query `N`. Local and neighbor result sets are unionized into a single result set by a unionizer operator `U` that must take the form of either `UNION` or `CONCAT`. A merge query `M` is applied that takes as input the result set and returns a new result set. The final result set is sent to the client, i.e. another node or an originator.

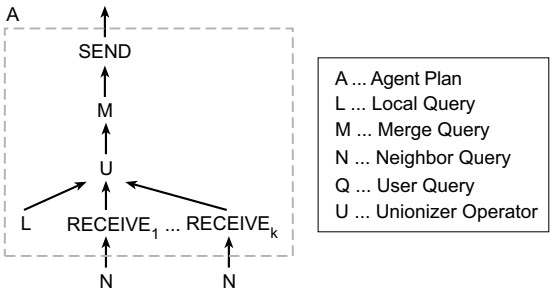

Fig. 3. Template Execution Plan.

Centralized Execution Plan. To see that indeed any query against any kind of database system can be answered within this framework we derive a simple *centralized execution plan* that always satisfies the semantics of any query `Q`. The plan substitutes specific subplans into the template plan `A`, leading to distinct plans for the agent node (Figure 4-a) and neighbors nodes (Figure 4-b). In the case of XQuery and SQL, parameters are substituted as follows:

XQuery	**SQL**
`A: M=Q, U=UNION, L="RETURN /", N'=N` `N: M=IDENTITY, U=UNION, L="RETURN /", N'=N`	`A: M=Q, U=UNION, L="SELECT *", N'=N` `N: M=IDENTITY, U=UNION, L="SELECT *", N'=N`

In other words, the agent's plan `A` fetches all raw tuples from the local and all remote databases, unionizes the result sets, and then applies the query `Q`. Neighbors are handed a rewritten neighbor query `N` that recursively fetches all raw tuples, and returns their union. The neighbor query `N` is recursively partitionable (see below).

The same centralized plan works for routed and direct response, both with and without metadata. Under direct response, a node does forward the query `N`, but does not attempt to receive remote result sets (conceptually empty result sets are delivered). The node does not send a result set to its predecessor, but directly back to the agent.

The centralized execution plan can be inefficient because potentially large amounts of base data have to be shipped to the agent before locally applying the user's query. However, sometimes this is the only plan that satisfies the semantics of a query. This is always the case for a complex query. A more efficient execution plan can sometimes be derived (as proposed below). This is always the case for a simple and medium query.

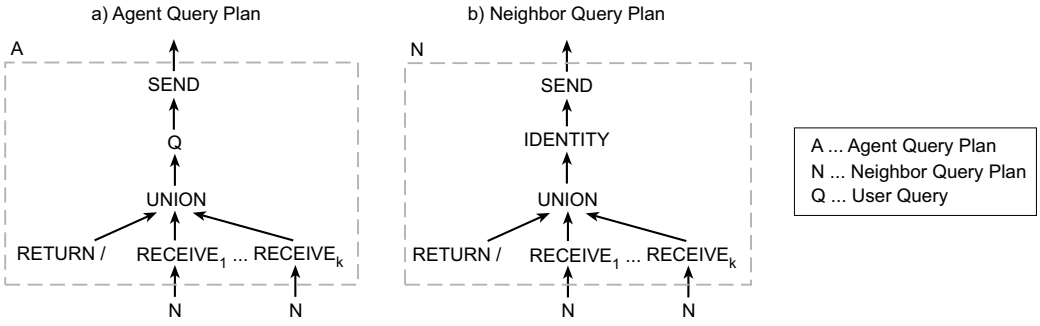

Fig. 4. Centralized Execution Plan.

Recursively Partitionable Query. A P2P network can be efficient in answering queries that are recursively partitionable. A query `Q` is *recursively partitionable* if, for the template plan `A`, there exists a merge query `M` and a unionizer operator `U` to satisfy the semantics of the query `Q` assuming that `L` and `N` are chosen as `L = Q` and `N = A`. In other words, a query is recursively partitionable if the very same execution plan *can* be recursively applied at every node in the P2P topology. The corresponding execution plan is depicted in Figure 5.

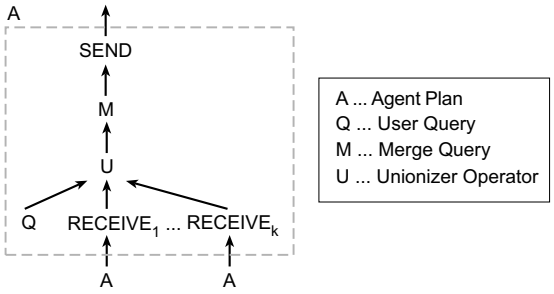

Fig. 5. Execution Plan for Recursively Partitionable Query.

The input and output of a merge query have the same form as the output of the local query `L`. Query processing can be parallelized and spread over all participating nodes. Potentially very large amounts of information can be searched while investing little resources such as processing time per individual node. The recursive parallel spread of load implied by a recursively partitionable query is the basis of the massive P2P scalability potential. However, query performance is not necessarily good, for example due to high network I/O costs.

Now we are in the position to clarify the definition of simple, medium and complex queries.

- *Simple Query.* A query is *simple* if it is recursively partitionable using `M = IDENTITY, U = UNION`. Examples are *(1) Find all (available) services. (2) Find all services that implement a replica catalog service interface and that CMS members are allowed to use, and that have an HTTP binding for the replica catalog operation "XML getPFNs(String LFN)".*

- *Medium Query.* A query is a *medium* query if it is not simple, but it is recursively partitionable. Examples are *(1) Return the number of replica catalog services. (2) Find the two CMS execution services with minimum and maximum CPU load and return their service descriptions and load.*

– *Complex Query.* A query is *complex* if it is not recursively partitionable. Examples are *(1) Find all (execution service, storage service) pairs where both services of a pair live within the same domain. (2) Find all domains that run more than one replica catalog with CMS as owner.*

For simplicity, in the remainder of this paper we assume that the user explicitly provides `M` and `U` along with a query `Q`. If `M` and `U` are not provided as part of a query to any given node, the node acts defensively by assuming that the query is not recursively partitionable. Choosing `M` and `U` is straightforward for a human being. Consider for example the following medium XQueries.

– *Return the number of replica catalog services.* The merge query computes the sum of a set of numbers. The unionizer is CONCAT.

```
Q = RETURN <tuple>
              count(/tupleset/tuple/content/service[interface/@type="repcat"])
           </tuple>
M = RETURN <tuple> sum(/tupleset/tuple) </tuple>
U = CONCAT
```

– *Find the service with the largest uptime.*

```
Q=M= RETURN (/tupleset/tuple[@type="service"] SORTBY (./@uptime)) [last()]
U =  UNION
```

Note that the query engine always encapsulates the query output with a `tupleset` root element. A query need not generate this root element as it is implicitly added by the environment.

Pipelining. The success of many applications depends on how fast they can start producing initial/relevant portions of the result set rather than how fast the entire result set is produced [33]. Often an originator would be happy to already do useful work with one or a few *early results*, as long as they arrive quickly and reliably. Results that arrive later can be handled later, or are ignored anyway. This is particularly often the case in distributed systems where many nodes are involved in query processing, each of which may be unresponsive for many reasons. The situation is even more pronounced in systems with loosely coupled autonomous nodes.

Operators of any kind have a uniform iterator interface, namely the three methods `open()`, `next()` and `close()`. For efficiency, the method `next()` can be asked to deliver several results at once in a so-called *batch*. Semantics are as follows: *"Give me a batch of at least* `N` *and at most* `M` *results"* (less than `N` results are delivered when the entire query result set is exhausted). For example, the SEND and RECEIVE network communication operators typically work in batches.

The monotonic semantics of certain operators such as SELECT, UNION, CONCAT, SEND, RECEIVE allow that operator implementations consume just one or a few child results on `next()`. In contrast, the non-monotonic semantics of operators such as SORT, GROUP, MAX, some JOIN methods, etc. require that operator implementations consume *all* child results already on `open()` in order to be able to deliver a result on the first call to `next()`. Since the output of these operators on a subset of the input is not, in general, a subset of the output on the whole input, these operators need to see all of their input before they produce the correct output. This does not break the iterator concept but has important latency and performance implications. Whether the root operator of an agent exhibits a short or long latency to deliver to the originator the first result from the result set depends on the query operators in use, which in turn depend on the given query. In other words, for some query types the originator has the potential to immediately start piping in results (at moderate performance rate), while for other query types it must wait for a long time until the first result becomes available (the full result set arrives almost at once, however).

A query (an operator implementation) is said to be *pipelined* if it can already produce at least one result tuple before all input tuples have been seen. Otherwise, a query (an operator) is said to be *non-pipelined*. Simple queries do support pipelining (e.g. Gnutella queries). Medium queries may or may not support pipelining, whereas complex queries typically do not support pipelining.

3 Static Loop Timeout and Dynamic Abort Timeout

Clearly there comes a time when a user is no longer interested in query results, no matter whether any more results might be available. The query roaming the network and its response traffic should fade away after some time. In addition, P2P systems are well advised to attempt to limit resource consumption by defending against *runaway* queries roaming forever or producing gigantic result sets, either unintended or malicious. To address these problems, an absolute *abort timeout* is attached to a query, as it travels across hops. An abort timeout can be seen as a deadline. Together with the query, a node tells a neighbor *"I will ignore (the rest of) your result set if I have not received it before 12:00:00 today."* The problem, then, is to ensure that a maximum of results can be delivered reliably within the time frame desired by a user. The value of a *static timeout* remains unchanged across hops, except for defensive modification in flight triggered by runaway query detection (e.g. infinite timeout). In contrast, it is intended that the value of a *dynamic timeout* be decreased at each hop. Nodes further away from the originator may time out earlier than nodes closer to the originator.

Dynamic Abort Timeout. A static abort timeout is entirely unsuitable for non-pipelined result set delivery, because it leads to a serious reliability problem, which we propose to call *simultaneous abort timeout.* If just one of the many nodes in the query path fails to be responsive for whatever reasons, all other nodes in the path are waiting, eventually time out and attempt to return at least a partial result set. However, it is impossible that any of these partial results ever reach the originator, because all nodes time out *simultaneously* (and it takes some time for results to flow back).

To address the simultaneous abort timeout problem, we propose dynamic abort timeouts. Under *dynamic abort timeout*, nodes further away from the originator time out earlier than nodes closer to the originator. This provides some safety time window for the partial results of any node to flow back across multiple hops to the originator. Intermediate nodes can and should adaptively decrease the timeout value as necessary, in order to leave a large enough time window for receiving and returning partial results subsequent to timeout.

Observe that the closer a node is to the originator, the more important it is (if it cannot meet its deadline, results from a large branch are discarded). Further, the closer a node is to the originator, the larger is its response and bandwidth consumption. Thus, as a good policy to choose the safety time window, we propose *exponential decay with halving.* The window size is halved at each hop, leaving large safety windows for important nodes and tiny window sizes for nodes that contribute only marginal result sets. Also, taking into account network latency and the time it takes for a query to be locally processed, the timeout is updated at each hop `N` according to the following recurrence formula:

$$timeout_N = currenttime_N + \frac{timeout_{N-1} - currenttime_N}{2} \tag{1}$$

Consider for example Figure 6. At time `t` the originator submits a query with a dynamic abort timeout of `t+4` seconds. In other words, it warns the agent to ignore results after time `t+4`. The agent in turn intends to safely meet the deadline and so figures that it needs to retain a safety window of 2 seconds, already starting to return its (partial) results at time `t+2`. The agent warns its own neighbors to ignore results after time `t+2`. The neighbors also intend to safely meet the deadline. From the 2 seconds available, they choose to allocate 1 second, and leave the rest to the branch remaining above. Eventually, the safety window becomes so small that a node can no longer meet a deadline on timeout. The results from the unlucky node are ignored, and its partial results are discarded. However, other nodes below and in other branches are unaffected. Their results survive and have enough time to hop all the way back to the originator before time `t+4`.

Static Loop Timeout. The same query may arrive at a node multiple times, along distinct routes, perhaps in a complex pattern. For reliable loop detection, a query has an identifier and a certain life

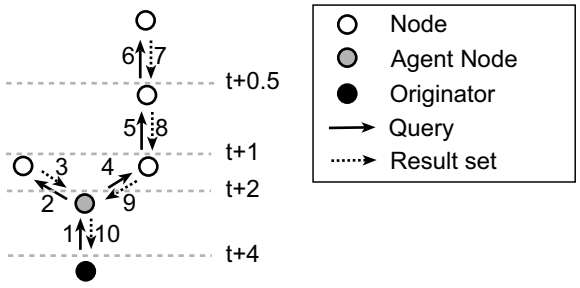

Fig. 6. Dynamic Abort Timeout.

time. To each query, an originator attaches a *loop timeout* and a different *transaction identifier*, which is a universally unique identifier (UUID). A node maintains a state table of transaction identifiers and returns an error when a query is received that has already been seen and has not yet timed out. On loop timeout, a node may "forget" about a query by deleting it from the state table. To be able to reliably detect a loop, a node must not forget a transaction identifier before its loop timeout has been reached. Interestingly, a static loop timeout is required in order to fully preserve query semantics. Otherwise, a problem arises that we propose to call *non-simultaneous loop timeout.* The non-simultaneous loop timeout problem is caused by the fact that some nodes still forward the query to other nodes when the destinations have already forgotten it. In other words, the problem is that loop timeout does not occur simultaneously everywhere. Consequently, a loop timeout must be static (does not change across hops) to guarantee that loops can reliably be detected. Along with a query, an originator not only provides a dynamic abort timeout, but also a static loop timeout. Initially at the originator, both values must be identical (e.g. `t+4`). After the first hop, both values become unrelated.

To summarize, we have `abort timeout` $\leq$ `loop timeout.` To ensure reliable loop detection, a loop timeout must be static whereas an abort timeout may be static or dynamic. Under non-pipelined result set delivery, dynamic abort timeout using *exponential decay with halving* ensure that a maximum of results can be delivered reliably within the time frame desired by a user. We speculate that dynamic timeouts could also incorporate sophisticated cost functions involving latency and bandwidth estimation and/or economic models.

4 Query Scope

As in a data integration system [8–10], the goal is to exploit several independent information sources as if they were a single source. This is important for distributed systems in which node topology or deployment model change frequently. For example, cross-organizational Grids and P2P networks exhibit such a character. However, in practice, it is often sufficient (and much more efficient) for a query to consider only a subset of all tuples (service descriptions) from a subset of nodes. For example, a typical query may only want to search tuples (services) within the scope of the domain `cern.ch` and ignore the rest of the world. To this end, we cleanly separate the concepts of (logical) query and (physical) query scope. A query is formulated against a global database view and is insensitive to link topology and deployment model. In other words, to a query the set of tuples appears as a single homogenous database, even though the set may be (recursively) partitioned across many nodes and databases. This means that in a relational or XML environment, at the global level, the set of all tuples appears as a single, very large, table or XML document, respectively. The query scope, on the other hand, is used to navigate and prune the link topology and filter on attributes of the deployment model. Conceptually, the scope is the input fed to the query. The query scope is a set and may contain anything from all tuples in the universe to none. Both query and scope can prune the search space,

but they do so in a very different manner. A query scope is specified either *directly* or *indirectly*. One can distinguish scopes based on neighbor selection, timeout and radius.

Neighbor Selection. For simplicity, all our discussions so far have implicitly assumed a *broadcast* model (on top of TCP) in which a node forwards a query to all neighbor nodes. However, in general one can select a subset of neighbors, and forward concurrently or sequentially. Fewer query forwards lead to less overall resource consumption. The issue is critical due to the snowballing (epidemic, flooding) effect implied by broadcasting. Overall bandwidth consumption grows exponentially with the query radius, producing enormous stress on the network and drastically limiting its scalability [34, 35].

Clearly selecting a neighbor subset can lead to incomplete coverage, missing important results. The best policy to adopt depends on the context of the query and the topology. For example, the scope can select only neighbors with a service description of interface type "Gnutella". In an attempt to explicitly exploit topology characteristics, a virtual organization of a Grid may deliberately organize global, intermediate and local job schedulers into a tree-like topology. Correct operation of scheduling may require reliable discovery of all or at least most relevant schedulers in the tree. In such a scenario, random selection of half of the neighbors at each node is certainly undesirable. A policy that selects all `child` nodes and ignores all `parent` nodes may be more adequate. Further, a node may maintain statistics about its neighbors. One may only select neighbors that meet minimum requirements in terms of latency, bandwidth or historic query outcomes (`maxLatency, minBandwidth, minHistoricResult`). Other node properties such as hostname, domain name, owner, etc. can be exploited in query scope guidance, for example to implement security policies. Consider an example where the scheduling system may only trust nodes from a select number of security domains. Here a query should never be forwarded to nodes not matching the trust pattern.

Further, in some systems, finding a single result is sufficient. In general, a user or any given node can guard against unnecessarily large result sets, message sizes and resource consumption by specifying the maximum number of result tuples (`maxResults`) and bytes (`maxResultsBytes`) to be returned. Using sequential propagation, depending on the number of results already obtained from the local database and a subset of the selected neighbors, the query may no longer need to be forwarded to the rest of the selected neighbors.

Neighbor Selection Query. For flexibility and expressiveness, we propose to allow the user to specify the selection policy. In addition to the normal query, the user defines a *neighbor selection query* (XQuery) that takes the tuple set of the current node as input and returns a subset that indicates the nodes selected for forwarding. For example, a neighbor query implementing broadcasting selects all services with registry and P2P query capabilities, as follows:

```
RETURN /tupleset/tuple[@type="service"
  AND content/service/interface[@type="Consumer-1.0"]
  AND content/service/interface[@type="XQuery-1.0"]]
```

A wide range of policies can be implemented in this manner. The neighbor selection policy can draw from the rich set of information contained in the tuples published to the node. Further, recall that the set of tuples in a database may not only contain service descriptions of neighbor nodes (e.g. in WSDL [19]), but also other kind of (soft state) content published from any kind of content provider. For example, this may include the type of queries neighbor nodes can answer, descriptions of the kind of tuples they hold (e.g. their types), or a compact summary or index of their content. Content available to the neighbor selection query may also include host and network information as well as statistics that a node periodically publishes to its immediate neighbors. A neighbor selection query enables group communication to all nodes with certain characteristics (e.g. the same group ID or interfaces). One can implement domain filters and security filters (e.g. `allow/deny` regular expressions as used in the Apache HTTP server if the tuple set includes metadata such as hostname and node owner. To summarize, a neighbor selection query can be used to implement *smart dynamic routing*.

Radius. The *radius* of a query is a measure of path length. More precisely, it is the maximum number of hops a query is allowed to travel on any given path. The radius is decreased by one at each hop. The roaming query and response traffic must fade away upon reaching a radius of less than zero. A scope based on radius serves similar purposes as a timeout. Nevertheless, timeout and radius are complementary scope features. The radius can be used to indirectly limit result set size. In addition, it helps to limit latency and bandwidth consumption and to guard against runaway queries with infinite lifetime. In Gnutella and Freenet, the radius is the primary means to specify a query scope. The radius is termed *TTL (time-to-live)* in these systems. Neither of these systems support timeouts.

For maximum result set size limiting, a timeout and/or radius can be used in conjunction with neighbor selection, routed response, and perhaps sequential forward, to implement the *expanding ring* [36] strategy. The term stems from IP multicasting. Here an agent first forwards the query to a small radius/timeout. Unless enough results are found, the agent forwards the query again with increasingly large radius/timeout values to reach further into the network, at the expense of increasingly large overall resource consumption. On each expansion radius/timeout are multiplied by some factor.

5 Related Work

Pipelining. For a survey of adaptive query processing, including pipelining, see the special issue of [37]. [38] develops a general framework for producing partial results for queries involving any non-monotonic operator. The approach inserts update and delete directives into the output stream. The Tukwila [39] and Niagara projects [40] introduce data integration systems with adaptive query processing and XML query operator implementations that efficiently support pipelining. Pipelining of hash joins is discussed in [41–43]. Pipelining is often also termed *streaming* or *non-blocking* execution.

Neighbor Selection. *Iterative deepening* [44] is a similar technique to *expanding ring* where an optimization is suggested that avoids reevaluating the query at nodes that have already done so in previous iterations. Neighbor selection policies that are based on randomness and/or historical information about the result set size of prior queries are simulated and analyzed in [45]. An efficient neighbor selection policy is applicable to simple queries posed to networks in which the number of links of nodes exhibits a power law distribution (e.g. Freenet and Gnutella) [46]. Here most (but not all) matching results can be reached with few hops by selecting just a very small subset of neighbors (the neighbors that themselves have the most neighbors to the n-th radius). Note, however, that the policy is based on the assumption that not all results must be found and that all query results are equally relevant. These related works discuss in isolation neighbor selection techniques for a particular query type, without the context of a framework for comprehensive query support.

JXTA. The goal of the JXTA P2P network [47–49] is to have peers that can cooperate to form self-organized and self-configured peer groups independent of their position in the network, and without the need of a centralized management infrastructure. JXTA defines six stateless best-effort protocols for ad hoc, pervasive, and multi-hop P2P computing. These are designed to run over uni-directional, unreliable transports. Due to this ambitious goal, a range of well-known higher level abstractions (e.g. bi-directional secure messaging) are (re)invented from first principles.

The Endpoint Routing Protocol allows to discover a route (sequence of hops) from one peer to another peer, given the destination peer ID. The Rendezvous Protocol offers publish-subscribe functionality within a peer group. The Peer Resolver Protocol and Peer Discovery Protocol allow for publication of advertizements and *simple* queries that are unreliable, stateless, non-pipelined, and non-transactional. We believe that this limits scalability, efficiency and applicability for service discovery and other non-trivial use cases. Lacking expressive means for query scoping, neighbor selection and timeouts, it is unclear how chained rendezvous peers can form a search network. We believe that JXTA

Peer Groups, JXTA search and publish/subscribe can be expressed within our UPDF framework, but not vice versa.

DNS. Distributed databases with a hierarchical name space such as the Domain Name System (DNS) [50] can efficiently answer *simple* queries of the form *"Find an object by its full name"*. Queries are not forwarded (routed) through the (hierarchical) link topology. Instead, a node returns a *referral* message that redirects an originator to the next closer node. The originator explicitly queries the next node, is referred to yet another closer node, and so on. To support neighbor selection in a hierarchical name space within our UPDF framework, a node could publish to its neighbors not only its service link, but also the name space it manages. The DNS referral behavior can be implemented within UPDF by using a radius scope of zero. The same holds for the LDAP referral behavior (see below).

X.500, LDAP and MDS. The hierarchical distributed X.500 directory [51] works similarly to the DNS. It also supports referrals, but in addition can forward queries through the topology (*chaining* in X.500 terminology). The query language is simple [3]. Query scope specification can support maximum result set size limiting. It does not support radius and dynamic abort timeout as well as pipelined query execution across nodes. LDAP [29] is a simplified subset of X.500. Like DNS, it supports referrals but not query forwarding. The Metacomputing Directory Service (MDS) [30] inherits all properties of LDAP. MDS additionally implements a simple form of query forwarding that allows for multi-level hierarchies but not for arbitrary topologies. Here neighbor selection forwards the query to LDAP servers overlapping with the query name space. The query is forwarded "as is", without loop detection. Further, MDS does not support radius and dynamic abort timeout, pipelined query execution across nodes as well as direct response and metadata responses.

6 Conclusions

Traditional distributed systems assume a particular type of topology (e.g. hierarchical as in DNS, LDAP). Existing P2P systems are built for a single application and data type and do not support queries from a general-purpose query language. For example, Gnutella, Freenet, Tapestry, Chord, Globe and DNS only support lookup by key (e.g. globally unique name). Others such as SDS, LDAP and MDS support simple special-purpose query languages, leading to special-purpose solutions unsuitable for multi-purpose service and resource discovery in large heterogeneous distributed systems spanning many administrative domains. LDAP and MDS do not support essential features for P2P systems such as reliable loop detection, non-hierarchical topologies, dynamic abort timeout, query pipelining across nodes as well as radius scoping. None introduce a unified P2P database framework for general-purpose query support.

We propose the *Unified Peer-to-Peer Database Framework (UPDF)* for general-purpose query support in large heterogeneous distributed systems spanning many administrative domains. UPDF is unified in the sense that it allows to express specific applications for arbitrary query languages (e.g. XQuery, SQL) and node topologies, and a wide range of data types, query response modes (e.g. Routed, Direct and Referral Response), neighbor selection policies, pipelining characteristics, timeout and other scope options. The uniformity, wide applicability and reusability of our approach distinguish it from related work, which individually addresses some but not all problem areas.

We are starting to build a system prototype with the aim of reporting on experience gained from application to an existing large distributed system such as the European Data Grid.

References

1. Ian Foster, Carl Kesselman, and Steve Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *Int'l. Journal of Supercomputer Applications*, 15(3), 2001.

2. Large Hadron Collider Committee. Report of the LHC Computing Review. Technical report, CERN/LHCC/2001-004, April 2001. http://cern.ch/lhc-computing-review-public/Public/Report_final.PDF.
3. Wolfgang Hoschek. *A Unified Peer-to-Peer Database Framework for XQueries over Dynamic Distributed Content and its Application for Scalable Service Discovery*. PhD Thesis, Technical University of Vienna, March 2002.
4. Ben Segal. Grid Computing: The European Data Grid Project. In *IEEE Nuclear Science Symposium and Medical Imaging Conference*, Lyon, France, October 2000.
5. Wolfgang Hoschek, Javier Jaen-Martinez, Asad Samar, Heinz Stockinger, and Kurt Stockinger. Data Management in an International Data Grid Project. In *1st IEEE/ACM Int'l. Workshop on Grid Computing (Grid'2000)*, Bangalore, India, December 2000.
6. Dirk Düllmann, Wolfgang Hoschek, Javier Jean-Martinez, Asad Samar, Ben Segal, Heinz Stockinger, and Kurt Stockinger. Models for Replica Synchronisation and Consistency in a Data Grid. In *10th IEEE Symposium on High Performance and Distributed Computing (HPDC-10)*, San Francisco, California, August 2001.
7. Ian Foster, Carl Kesselman, Jeffrey Nick, and Steve Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration, January 2002. http://www.globus.org.
8. J.D. Ullman. Information integration using logical views. In *Int'l. Conf. on Database Theory (ICDT)*, Delphi, Greece, 1997.
9. Daniela Florescu, Ioana Manolescu, Donald Kossmann, and Florian Xhumari. Agora: Living with XML and Relational. In *Int'l. Conf. on Very Large Data Bases (VLDB)*, Cairo, Egypt, February 2000.
10. A. Tomasic, L. Raschid, and P. Valduriez. Scaling access to heterogeneous data sources with DISCO. *IEEE Transactions on Knowledge and Data Engineering*, 10(5):808–823, 1998.
11. M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, 1999.
12. Ann Chervenak, Ewa Deelman, Ian Foster, Leanne Guy, Wolfgang Hoschek, Adriana Iamnitchi, Carl Kesselman, Peter Kunszt, Matei Ripeanu, Bob Schwartzkopf, Heinz Stockinger, Kurt Stockinger, and Brian Tierney. Giggle: A Framework for Constructing Scalable Replica Location Services. In *Proc. of the Int'l. IEEE/ACM Supercomputing Conference (SC 2002)*, Baltimore, USA, November 2002. IEEE Computer Society Press.
13. Leanne Guy, Peter Kunszt, Erwin Laure, Heinz Stockinger, and Kurt Stockinger. Replica Management in Data Grids. Technical report, Global Grid Forum Informational Document, GGF5, Edinburgh, Scotland, July 2002.
14. Heinz Stockinger, Asad Samar, Shahzad Mufzaffar, and Flavia Donno. Grid Data Mirroring Package (GDMP). *Journal of Scientific Programming*, 2002.
15. William Bell, Diana Bosio, Wolfgang Hoschek, Peter Kunszt, Gavin McCance, and Mika Silander. Project Spitfire - Towards Grid Web Service Databases. Technical report, Global Grid Forum Informational Document, GGF5, Edinburgh, Scotland, July 2002.
16. Nelson Minar. Peer-to-Peer is Not Always Decentralized. In *The O'Reilly Peer-to-Peer and Web Services Conference*, Washington, D.C., November 2001.
17. Wolfgang Hoschek. The Web Service Discovery Architecture. In *Proc. of the Int'l. IEEE/ACM Supercomputing Conference (SC 2002)*, Baltimore, USA, November 2002. IEEE Computer Society Press.
18. Wolfgang Hoschek. A Database for Dynamic Distributed Content and its Application for Service and Resource Discovery. In *Int'l. IEEE Symposium on Parallel and Distributed Computing (ISPDC 2002)*, Iasi, Romania, July 2002.
19. E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1. *W3C Note 15*, 2001. http://www.w3.org/TR/wsdl.
20. World Wide Web Consortium. XQuery 1.0: An XML Query Language. *W3C Working Draft*, December 2001.
21. Wolfgang Hoschek. A Comparison of Peer-to-Peer Query Response Modes. In *Proc. of the Int'l. Conf. on Parallel and Distributed Computing and Systems (PDCS 2002)*, Cambridge, USA, November 2002.
22. Gnutella Community. Gnutella Protocol Specification v0.4. dss.clip2.com/GnutellaProtocol04.pdf.
23. I. Clarke, O. Sandberg, B. Wiley, and T. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Workshop on Design Issues in Anonymity and Unobservability*, 2000.
24. B. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-resilient wide-area location and routing. Technical report, U.C. Berkeley UCB//CSD-01-1141, 2001.

25. I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *ACM SIGCOMM*, 2001.
26. M. van Steen, P. Homburg, and A. Tanenbaum. A wide-area distributed system. *IEEE Concurrency*, 1999.
27. Steven E. Czerwinski, Ben Y. Zhao, Todd Hodes, Anthony D. Joseph, and Randy Katz. An Architecture for a Secure Service Discovery Service. In *Fifth Annual Int'l. Conf. on Mobile Computing and Networks (MobiCOM '99)*, Seattle, WA, August 1999.
28. J. Waldo. The Jini architecture for network-centric computing. *Communications of the ACM*, 42(7), July 1999.
29. W. Yeong, T. Howes, and S. Kille. Lightweight Directory Access Protocol. *IETF RFC 1777*, March 1995.
30. Karl Czajkowski, Steven Fitzgerald, Ian Foster, and Carl Kesselman. Grid Information Services for Distributed Resource Sharing. In *Tenth IEEE Int'l. Symposium on High-Performance Distributed Computing (HPDC-10)*, San Francisco, California, August 2001.
31. International Organization for Standardization (ISO). Information Technology-Database Language SQL. *Standard No. ISO/IEC 9075:1999*, 1999.
32. Donald Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys*, September 2000.
33. T. Urhan and M. Franklin. Dynamic Pipeline Scheduling for Improving Interactive Query Performance. *The Very Large Database (VLDB) Journal*, 2001.
34. Jordan Ritter. Why Gnutella Can't Scale. No, Really. http://www.tch.org/gnutella.html.
35. Matei Ripeanu. Peer-to-Peer Architecture Case Study: Gnutella Network. In *Int'l. Conf. on Peer-to-Peer Computing (P2P2001)*, Linkoping, Sweden, August 2001.
36. S.E. Deering. *Multicast Routing in a Datagram Internetwork*. PhD Thesis, Stanford University, 1991.
37. IEEE Computer Society. *Data Engineering Bulletin*, 23(2), June 2000.
38. Jayavel Shanmugasundaram, Kristin Tufte, David J. DeWitt, Jeffrey F. Naughton, and David Maier. Architecting a Network Query Engine for Producing Partial Results. In *WebDB 2000*, 2000.
39. Zachary G. Ives, Alon Y. Halevy, and Daniel S. Weld. Integrating Network-Bound XML Data. *IEEE Data Engineering Bulletin*, 24(2), 2001.
40. J. F. Naughton, D. J. DeWitt, D. Maier, A. Aboulnaga, J. Chen, L. Galanis, J. Kang, R. Krishnamurthy, Q. Luo, N. Prakash, R. Ramamurthy, J. Shanmugasundaram, F. Tian, K. Tufte, S. Viglas, Y. Wang, C. Zhang, B. Jackson, A. Gupta, and R. Chen. The Niagara Internet Query System. *IEEE Data Engineering Bulletin*, 24(2), 2001.
41. Annita N. Wilschut and Peter M. G. Apers. Dataflow query execution in a parallel main-memory environment. In *First Int'l. Conf. on Parallel and Distributed Information Systems*, December 1991.
42. Zachary G. Ives, Daniela Florescu, Marc T. Friedman, Alon Y. Levy, and Daniel S. Weld. An adaptive query execution system for data integration. In *ACM SIGMOD Conf. On Management of Data*, 1999.
43. Tolga Urhan and Michael J. Franklin. Xjoin, A reactively-scheduled pipelined join operator. *IEEE Data Engineering Bulletin*, 23(2), June 2000.
44. Beverly Yang and Hector Garcia-Molina. Efficient Search in Peer-to-Peer Networks. In *22nd Int'l. Conf. on Distributed Computing Systems*, Vienna, Austria, July 2002.
45. Adriana Iamnitchi and Ian Foster. On Fully Decentralized Resource Discovery in Grid Environments. In *Int'l. IEEE Workshop on Grid Computing*, Denver, Colorado, November 2001.
46. A. Puniyani B. Huberman L. Adamic, R. Lukose. Search in power-law networks. *Phys. Rev*, E(64), 2001.
47. Bernard Traversat, Mohamed Abdelaziz, Mike Duigou, Jean-Christophe Hugly, Eric Pouyoul, and Bill Yeager. Project JXTA Virtual Network, 2002. White Paper, http://www.jxta.org.
48. Steven Waterhouse. JXTA Search: Distributed Search for Distributed Networks, 2001. White Paper, http://www.jxta.org.
49. Project JXTA. JXTA v1.0 Protocols Specification, 2002. http://spec.jxta.org.
50. P. Mockapetris. Domain Names - Implementation and Specification. *IETF RFC 1035*, November 1987.
51. International Telecommunications Union. Recommendation X.500, Information technology – Open System Interconnection – The directory: Overview of concepts, models, and services. *ITU-T*, November 1995.