Scalable Distributed Aggregate Computations through Collaboration in Peer-to-Peer Systems

Leonidas Galanis
University of Wisconsin - Madison
1210 W Dayton St, Madison, WI 53706, USA
lgalanis@cs.wisc.edu

David J. DeWitt
University of Wisconsin - Madison
1210 W Dayton St, Madison, WI 53706, USA
dewitt@cs.wisc.edu

Abstract
Computing aggregates over distributed data sets constitutes an interesting class of distributed queries. Recent advances in peer-to-peer discovery of data sources and query processing techniques have made such queries feasible and potentially more frequent. The concurrent execution of multiple and often identical distributed aggregate queries can place a high burden on the data sources. This paper identifies the scalability bottlenecks that can arise in large peer-to-peer networks from the execution of large numbers of aggregate computations and proposes a solution. In our approach peers are assigned the role of aggregate computation maintainers, which leads to a substantial decrease in requests to the data sources and also avoids duplicate computation by the sites that submit identical aggregate queries. Moreover, a framework is presented that facilitates the collaboration of peers in maintaining aggregate query results. Experimental evaluation of our design demonstrates that it achieves very good performance and scales to thousands of peers.

1. Introduction
Peer-to-Peer (P2P) computing has gained both scientific and social importance recently due to the success of systems such as Freenet [8], Gnutella [12] and Napster [19]. The wide adoption of the early applications of this technology has prompted numerous new applications such as Kazaa [16] and has led research teams worldwide to formulate algorithms that make P2P systems scalable and efficient. Harnessing P2P technology has the potential to produce systems that combine good scalability with minimal infrastructure cost. P2P systems are designed to start out small and seamlessly evolve to very large distributed systems with thousands of participants.

The P2P computing paradigm has inspired many research projects to focus on a large variety of open problems. [23], [24], [29] and [35] provide the basis for low-level location services, otherwise known as Distributed Hash Tables (DHTs). [6], [18] and [25] illustrate how to use DHTs to build distributed file systems. [15] and [10] attempt to explore P2P technology to process complex queries in large distributed systems. The result has been the emergence of a concept known as data centric networking ([14] and [28]). The Internet started out as host-centric system in which research has focused on efficient ways to interconnect physical hosts. With the advent of P2P systems finding interesting data efficiently has become a major focus of research in the networking community.

P2P research forms a major part of the data centric Internet initiative and promises efficient and scalable data location and complex query processing over large distributed systems. Aggregate computations on data from distributed data sources constitute an important class of queries. P2P tools promise to make such queries feasible and therefore more frequent. If an aggregate computation is interesting to multiple peers in the network, the data sources participating in the computation can expect to receive the same query multiple times. Thus a new problem arises: The many-to-many query problem (M2M), which places scalability limits on query processing in P2P systems.

M2M is best illustrated with an example. Consider an application that brings together traders around the world in a large P2P commodity trading system without a centralized infrastructure. The need for such systems is real as illustrated by the P2P real estate project [2]. Traders post their sale and bid prices based on information obtained by querying each other. Typically, participants determine their asking price or bid after consulting the maximum bid and minimum sale price for a commodity across all traders. In the absence of a central server, a trader has to query all other traders in order to determine the maximum bid and the minimum sale price. Thus, if \( m \) sellers and \( n \) bidders are trading on a particular commodity, each trader has to answer \( m+n-1 \) identical queries. Furthermore, the total number of messages that must be exchanged among the participants and the total number of queries executed in the system is \( (m+n)(m+n-1) \). If a central server were used each participant would require only 2 queries to retrieve the minimum sale price and the maximum bid, which translates to \( (m+n) \) messages for 2 \( (m+n) \) queries and another \( (m+n) \) messages for the \( (m+n) \) updates of bids and sale prices on the central server. Consequently, any P2P system would still not scale well for this type of application due to the high message traffic and query processing load. On the other hand, a central server solution can scale by adding additional hardware. Maintaining a central infrastructure is certainly feasible but requires a business model and dedicated resources. A P2P solution, however, uses existing
infrastructure and scales automatically as new participants join and contribute resources. We believe that, as P2P systems become more widespread, problems similar to M2M will arise in other P2P applications.

This work presents a framework for efficiently processing many-to-many aggregate queries over large P2P networks in a scalable fashion. Our approach requires the same number of queries and messages as would be required with a centralized system by leveraging DHT technology and catalog services. The contributions of this work can be summarized as follows:

1. A method for defining the special handling of aggregate computations that follow the many-to-many query pattern.
2. An efficient query processing strategy that leverages existing P2P technology and allows for scalable processing of many-to-many aggregate queries.
3. An experimental validation of our approach that demonstrates its scalability potential and, at the same time, shows the adverse impact of the M2M query problem on P2P applications.

We believe that our design opens up new possibilities for novel distributed applications.

Aggregation is going to be increasingly essential for efficiently surveying large amounts of distributed data in P2P networks. Thus, we expect various applications to require efficient aggregation over distributed data sources. The case study in this paper is a commodity trading application in which traders need timely information about maximum bids and minimum sale prices in order to define their strategy (Section 5). The P2P Real Estate application ([2]) is intended for searching listings that are stored on the personal databases of various realtors. One can easily conceive its extension to facilitate interesting aggregate computations based on various real estate features. Another application that will require distributed aggregations is the wide area network monitoring application outlined in [15]. Being able to obtain timely information about which points in the network have the lowest or highest hourly average traffic can aid in effective network management and assist in detecting intrusions. Comparison shopping is yet another area which will benefit from a distributed aggregation framework. A P2P system using our framework can provide timely price listings for a very large selection of products.

The paper is organized as follows: Section 2 states the problem definition. Section 3 outlines the overall system architecture and Section 4 delves into the detailed design of the distributed aggregated computation layer. Section 5 introduces the distributed application used in the experimental evaluation. Section 6 presents the results of the experiments. The paper ends with related work (Section 6.5) and concluding remarks (Section 8).

2. Problem Definition

Peer-to-peer query processing systems may experience scalability problems under specific query loads. The many-to-many query problem, which can arise in distributed aggregate computations, is discussed next. Consider a P2P network with a large number of aggregation queries that process data from multiple sites (the roles of data source and query origin need not be distinct, a feature common in P2P systems). These queries are interesting for many sites and so multiple sites submit these queries. Possible aggregation functions are the minimum, the maximum, the average, top(k) and others. Moreover, assume that the aggregation queries always need to fetch data from the data sources in order to compute the result accurately. Let Q be an aggregation query that requires data from a data sources in set S. Query Q is issued at an average rate r by b peers in set O. As the P2P network grows O and S will also grow. This approach leads to the scalability bottlenecks described next.

Each o, Δ O issues r a queries per time unit to collect the results needed to compute Q. During the same time period each s, Δ S has to process r b identical requests. This yields a total of 2 r a b network messages to carry requests and responses. This approach will not scale as a and b grow. Furthermore, if we assume that each request to a source s requires t time units in isolation, having b outstanding requests on average means that the average time for servicing each request by each o, will increase to approximately T = b t. Clearly, if the number of peers that attempt to compute query Q grows, the quality of service will degrade linearly. Finally, if one considers the fact that a very large number of different queries like Q can be outstanding at any point in time, one is forced to conclude that a pure peer-to-peer substrate is not a good design for distributed aggregate computations. The reason is that as O and S are expected to grow linearly, the number of network messages will grow quadratically, which will limit scalability.

In search of a better solution, one could employ a centralized design. Each query origin o, Δ O now issues only one query request to the central server C. Each data source s, Δ S maintains its data on C. The total number of messages is analogous to r (a + b), which is significantly better than the fully distributed case since it grows linearly. However, the quality of service in this case is dependent on the processing capacity of C. As the number of participating sites grows, the processing capacity of C has to be constantly upgraded in order to meet the response time requirements. Consequently, to achieve continued scaling, participants have to agree to constantly manage and fund C.

The question that immediately arises is whether one can achieve the desirable traits of the centralized solution within a pure P2P layer. The solution is to somehow decentralize the work of the central server C by utilizing the processing power contributed by the sites that join the P2P system. The straightforward method for distributing the work performed by C is to partition it by the aggregate computations. More specifically, each aggregate computation Q can be assigned to one of the peers in the system. Each peer acts as a hub for Q, receiving updates from the data sources and queries. This paper shows that the proposed setup works very well and produces scalable solu-
3. System Architecture

The architecture for each peer (node) on the network is depicted in Figure 1. The software on each peer consists of four layers: 1) a Distributed Hash Table layer (DHT), 2) the Catalog Service (CS) layer, 3) the Aggregate Computation (ACL) layer, and 4) a query engine with access to the local data. The aggregate computation layer is responsible for solving the M2M problem. Users are clients of the query engine. The design does not dictate a specific data model or query language, but all examples will assume XML [33] data sources and XPath [32] queries.

3.1 The Distributed Hash Table Layer

The DHT layer is based on existing technology ([23], [24], [29], [35]). Its purpose is to support the efficient and scalable location of keys or object identifiers used by the higher-level layers of the system. In essence, DHTs are fully distributed hash tables that employ protocols for efficiently directing requests for specific keys to the nodes in the network that are responsible for those keys. Responsibility for a key usually means storing an object that is uniquely associated with the key (e.g., a video clip). DHTs are robust with respect to node arrivals and departures and provide the application layer with replication facilities ([7]) that handle unexpected node failures. Typically the number of messages required in a network of \( n \) nodes is in the order of \( O(\log n) \), which makes DHTs highly scalable.

3.2 The Catalog Service Layer

The Catalog Service (CS) layer is the data discovery tool. While the DHT layer is ideal for routing key/object lookups to peers, it cannot be used for directing complex queries (such as XPath queries) to the relevant data sources. Therefore, each node employs a CS that, when given an arbitrary query, locates the relevant data sources. The CS is based on the framework presented in [10].

The basic functionality of the catalog service layer is best described using an example. Consider an XPath query \( q/\text{trades}/*/\text{bidder[items/item/category="collectibles"]} \) that retrieves all bidders that are currently interested in “collectible” items. Using the path structure and the equality predicate, the catalog service will produce a list of the peers with bidders of “collectible” items. Thus \( q \) will only be sent to the appropriate subset of nodes. More specifically, the XML element \( \text{category} \) is chosen to produce the DHT key. The DHT layer then determines the node \( N \) that is responsible for this key. \( N \) holds structural and value information provided by all the peers in the network that contain the element \( \text{category} \) in their data. \( N \) receives \( q \) and determines the set of nodes that will have a non-empty result set for \( q \).

To provide this functionality, the CS requires that each data source provide a summary of its data in a special form when it joins the system. For XML data sets, each element and attribute is associated with a structural summary and an optional data summary, both of which comprise the catalog information. For example, a node whose collection of XML documents contains the \( \text{category} \) element creates catalog information that contains the following: 1) a list of all paths that lead to \( \text{category} \) and 2) an optional summary of the values of \( \text{category} \), which further helps distinguish among a potentially large number of nodes that serve different categories. Using the DHT layer this catalog information is then assigned to a specific peer that is determined using the element or attribute. Note that each peer acts both as a data source and as a CS provider.

![Figure 1 System Architecture](image)

3.3 The Aggregate Computation Layer

The Aggregate Computation Layer (ACL) maintains the registered aggregate queries that have been submitted by the various nodes in the P2P network. It provides an interface for defining an aggregation point (AP) that corresponds to the registration of an aggregate computation \( Q \). \( Q \) is assigned to one or more peers that become responsible for maintaining the aggregate computation by receiving updates from nodes with data relevant to \( Q \). These peers also respond to queries from other peers. After an AP has been installed, the ACL on each node redirects queries and updates to the peers that take part in the maintenance of the aggregate computation.

Note that the ACL is tightly coupled with the Catalog Service and makes direct use of the catalog index structures. The ACL is the focus of this paper and its detailed design is presented in Section 4.

3.4 The Query Engine

The Query Engine is the most generic part of the system. It provides query and update functionality to the data on the site and uses the Catalog Service to distribute queries to the relevant nodes in the P2P network. The data itself may reside in a DBMS or in files. The site creates catalog information and provides an XPath query interface.
4. Distributed Aggregation

The Aggregate Computation Layer facilitates the creation of aggregation points for queries. This section presents the interface for installing aggregation points and describes the relevant mechanisms.

4.1 Aggregation Point Creation

Generally any querying site can request the creation of an aggregation point if it discovers that it frequently needs to contact a large number of nodes in order to compute an aggregate. Similarly a data source can request an AP if it becomes overwhelmed with large numbers of identical requests that are part of an aggregation computation. The ACL creates an aggregation point when provided with an activation record (AR) that contains the following fields:

**Aggregate Function:** This field determines the aggregate function (average, minimum, maximum etc) that is applied by the peer responsible for the aggregation computation maintenance to the incoming data.

**Target Data:** This field contains a query that defines the data needed for computing the aggregation. The target element or attribute of the query determines the catalog service peer ([10]) that is the peer that will create and maintain the AR.

**Scope:** The scope can be either global or local. Global scope means that the aggregate function should be computed over all peers with relevant data, while local scope computes one value for each peer.

**Group By:** The group by field refines the aggregation by defining aggregate groups. Essentially it is similar to the SQL “group by” construct. Each group is assigned to a node that maintains the aggregate computation for the specific group. The field can contain multiple group-by expressions.

<table>
<thead>
<tr>
<th>Aggregate</th>
<th>MAXIMUM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target Data</td>
<td>//bidder/item/current_bid</td>
</tr>
<tr>
<td>Scope</td>
<td>GLOBAL</td>
</tr>
<tr>
<td>Group By</td>
<td>//bidder/item/@item_id</td>
</tr>
</tbody>
</table>

An aggregation point corresponds to one or more groups depending on the group by field. Each group is assigned to a node, the Aggregation Point Host (APH). The APH is selected among the peers in the P2P system based on the activation record. A DHT key is computed using the target data and group by fields. This key determines which peer will currently serve as the APH and thus assume responsibility for the aggregate computation.

To illustrate the definition of an activation record, consider again the example of commodity traders. Suppose bidders are peers that post their bids online as XML documents and update them as they trade. Suppose that the current price a bidder is willing to pay for a commodity item i can be accessed using the path p_{bid} = //bidder/item[@item_id = i]/current_bid. Sellers naturally want to find the bidder with the maximum bid. Thus, if there are a large number of sellers of commodity i, and a large number of bidders, an aggregation point is needed to make trading more efficient. Table 1 shows what the required activation record looks like. The aggregate function is the maximum; the scope is global since a seller needs the peer with the currently highest bid and not the highest bid on each peer. The target path is p_{target} = //bidder/item/current_bid. Hence, current_bid is the catalog key that determines the peer that maintains the AR. The group by field is the path p_{group_by}(x) = //bidder/item/@item_id = x. The p_{group_by} field essentially assigns each traded item to a different DHT key.

After the installation of the aggregation point, sellers can determine the maximum bid, as well as the peer that hosts the maximum bidder, by querying the corresponding APH.

4.2 Redirection of Queries and Updates

This section presents the data structures that are maintained and the mechanisms that are used to redirect aggregation queries to the appropriate peers. Consider the aggregation point AP with an activation record AR_{AP} (Table 1). Let also AR_{i,AP}(target data) denote the relevant field in the activation record. AR_{i,AP}(target data) is, in essence, an XPath query that can be submitted to the catalog service for retrieving the relevant data sources. Let k = current_bid. Under normal operation the catalog service would use k to identify the relevant data sources. However, if an AP is installed, the mode of operation changes as described below.

Each node uses the Aggregation Key Map (AKM), which acts as a subscription system that associates catalog keys with their activation records. Figure 2 outlines the process based on the example from Section 4.1. Suppose that peer N_s, which hosts sellers of commodity i decides to request the creation of the aggregation point AP that is defined by the activation record described in Table 1. The request will be forwarded to the peer N_c that holds catalog information for the key current_bid (step 1). N_c will generate the association a_{current_bid} = current_bid \[ MAX_{GLOBAL}(p_{target}, p_{group_by}(x)) \] and insert it in the AKM. Note that multiple associations for a given key can exist in the AKM and they are selected based on the actual query. Then, N_c forwards a_{current_bid} to all peers N_h_k which host bidders and to N_s (step 2). N_c knows about all such peers since it is hosting catalog information for current_bid. Upon receipt of a_{current_bid} each N_h_k is expected to send updates about the current bid of its bidders for all traded commodities to specific other peers that form the set of aggregation point hosts (APH) (step 3). The DHT layer, using the information in a_{current_bid} can uniquely determine each peer in APH. For example, for commodity item i, the APH P_{AP,i} is determined by hashing the list (MAX, GLOBAL, p_{target}, p_{group_by(i)}) to retrieve a DHT key k_i. Thus P_{AP,i} becomes the maintainer of the requested aggregation for commodity i. Aggregation responsibility is tied to the key k_i and not to the peer P_{AP,i}. This way if P_{AP,i} leaves, the DHT ensures that k_i points to a different peer that will automatically become responsible for the maintenance of the aggregation.
Figure 2 Installation of an Aggregation Point

The question that remains is how other peers that host sellers other than N_s find out about the new aggregation points. This turns out to be straightforward since N_c is the designated node for all inquiries regarding current bid (Figure 3). Thus, any other node N_x which is the origin of a query such as Q_x = MAX([@item_id = y]/current_bid) will request catalog information from N_c (step 1). N_c will then provide N_x with the association a_current_bid (step 2). Having this information N_x can identify P_Ap_x as the peer to visit in order to obtain the answer to Q_x (step 3). Furthermore, N_x caches a_current_bid in its own AKM and thus only needs to contact N_c once.

Figure 3 Redirection of query Q_x to node P_Ap_x

4.3 Load Balancing

The mechanisms for distributed aggregation presented so far, essentially assign the task of maintaining an aggregate computation and answering relevant query requests to some peer in the network. If the assigned aggregate computation is very popular or is updated at a very high rate, the aggregate maintenance can overload the APH. For example popular commodities will involve many traders and consequently the rate of sale price and bid updates as well as the rate of queries for the maximum bid or minimum sale price will be significantly higher than those of other less popular commodities.

Replication as described in [10] can be readily applied to Aggregation Points, in order to distribute the query load. However, the rate in a M2M environment can also create performance problems. Therefore, we propose a method that deals with high request rates for both queries and updates.

4.3.1 Load Balancing Trigger

A peer P triggers a load balancing action when its queue L of outstanding requests exceeds some preset length L_t. When this occurs, the outstanding requests are examined and the n most frequent in L are chosen for load balancing.

If the request is an aggregation layer request the following takes place. Let the key, which is computed from the activation record AR be k. To redistribute the load for k the peer creates a set of new keys K, and records the association m_k = k [] K in a separate structure, the meta-key map. Subsequently other peers that request aggregate results using k from P receive m_k, cache it in their own meta-key map, and use it without contacting P again. The meta-key map is consulted after the AKM so as to find out if the key of an aggregation is subject to load balancing. The size |K| of set K need not be specific, since as it will be shown later more keys can be created for k as required. Figure 4 shows all the stages required to determine a DHT key given an arbitrary query. Next we present how redirection of queries and requests takes place for load-balanced aggregation keys.

Figure 4 Key determination pipeline on a peer

4.3.2 Update and Query Load Balancing

Data sources that receive the association m_k and need to send updates for key k use the set K to send updates in a round-robin fashion starting from a random key. Hence, the update load is distributed approximately equally among the keys in K. The peers that correspond to the keys in K - peers(K) - maintain a partial aggregate. Computing the actual aggregate from the partial aggregates is straightforward for the common aggregate functions and is interchangeably assigned to a peer in peers(K), the master peer.

The master assignment needs to cycle through all elements in peers(K) so as to distribute the query load evenly over time. Additionally, the method for choosing the master peer P_M must be such that all the peers in the network can produce P_M solely from the set K without the need of global message exchange. Therefore, we select P_M using K based on the current time, which is possible given existing network services like the Network Time Protocol [20], which allows clock synchronization with an accuracy of several milliseconds. Thus, the current master key is obtained as the f(t)^th element of K where f(t) = (t/T) modulo |K|, where t is the current time and T is the period during which a given peer acts as the master. Consequently, all keys in K receive equal master time. When a query arrives at a peer that is not a master anymore it is forward to the current master by re-computing f.

Updates from the nodes in peers(K = {P_M}) are forwarded to P_M only if the respective partial aggregates change. This means that P_M will generally not experience the total update rate targeted at key k. Note, nevertheless, that the computed aggregate found on P_M arrives there using two network messages when a partial aggregate has changed and one otherwise. Without load balancing there is always one network message: from the data source to the
aggregation point peer. We believe that the additional message is not a problem since load balancing will speed up data processing, in such a way that the possible extra network delay to reach \( P_M \) will not be significant. During a master transition, the old master sends the current value of the aggregate to the new one. Subsequently, all query and update requests for a given aggregate are directed to the current master \( P_M \), which holds the most current value of the aggregate. Thus, nodes in peers\( (K) \) work in a round-robin fashion experiencing periodic request peaks that would otherwise be endured by the original peer \( P \) that initiated the load balancing action, which would lead to a large backlog.

### 4.3.3 Cascading Load Balancing

If a node in peers\( (K) \) experiences a load the load balancing action of \( P_M \) is made less sensitive to query load since it is temporarily experiencing the same request rate that led to the original load balancing action. Thus, the master initiates a load balancing action only after queue \( L \) becomes longer than \( m/H \) (for some small integer \( m \)). In both cases a new association is created in the meta-key map and points to the existing set \( K \), which is augmented with additional keys.

#### 4.4 Data Structure Sizes

The ACL is tightly coupled with the Catalog Service and makes direct use of the catalog index structures. The aggregates are numbers stored next to the data summaries of their corresponding path. For example, if an aggregation point has \( m \) groups, \( m \) numbers along with their corresponding paths will be distributed across the peers. Furthermore, the AKM and the meta-key map need only cache associations that are relevant to a peer’s workload. When the workload changes, the necessary associations can be retrieved from the peer that holds the relevant catalog information. Hence, the storage requirements for the ACL and the load balancing mechanism are small.

#### 4.5 Aggregation Computation Accuracy

Computing a transaction-consistent aggregate value in a distributed environment would require imposing distributed transactions on data sources. We believe that this is unrealistic objective in a network of autonomous and independent data sources. Therefore, in a distributed environment without globally enforceable transactions the only way to increase the accuracy of aggregate computations is to improve their response time, by decreasing the time it takes to collect all the relevant data. The aggregation point framework eases the burden of data sources affected by the M2M problem by providing concentration points for aggregate computations in the form of Aggregation Point Hosts. The APHs do not store the actual data but just maintain the aggregate computation, which is an easier task. Furthermore, using load balancing techniques described in this paper and \([10]\), the load of an APH can be shared with other peers in the network. Our experiments in Section 6.4 show that the Aggregation Point framework leads to higher throughput and better response time and so to timelier and more accurate aggregate computations. Finally, the aggregation result is more consistent since it does not depend on the timing of distributed data collection.

### 5. Case Study

Our work focuses on the scalability aspect of distributed aggregate computations in a P2P environment, which is a step towards building the distributed query processing systems of the future. We evaluated our design using a distributed commodity trading scenario introduced next.

#### 5.1 Distributed Commodity Trading

Regular commodity trading involves multiple bidders and sellers all of whom are trading on a specific commodity. Trading is carried out in a centralized manner where an auctioneer determines the way sellers and bidders are paired up for the realization of sales. Each seller submits an asking price to the auctioneer and each bidder submits a bid. The auctioneer pairs up sellers and bidders using some algorithm and then “shouts” out the prices of the transactions. Sellers and bidders use the price information to determine their bids and selling prices for the next round. Note that since a specific commodity type \( (item) \) can be on sale by more than one seller, a bidder need not bid the maximum price in order to buy a commodity. Furthermore, there is no auction ending time and so no trading “spikes” as in regular auctions with one seller and many bidders. Trading is continuous and ends when there are no more items to sell.

The elimination of both the central infrastructure and the fixed time rounds is a motivation to realize Distributed Commodity Trading (DCT). In DCT each seller searches for the bidder with the currently maximum bid. If it is above the selling price, the transaction proceeds between the seller and the bidder without the intervention of an intermediary. At the same time bidders will issue similar queries to the system in order to modify their bids based on market information. Our framework provides traders with the means to discover and query each other and with timely information that allows them to define their strategy. The evaluation shows that our design can scale to very large networks of traders.

### 5.2 Application Scenario

#### 5.2.1 Trader Behavior

Two kinds of traders take part in simulated trading: Bidders and Sellers as described in \([21]\). Sellers possess items available for sale for which bidders are interested. Sellers perform specific tasks for each item they have for sale. First, a seller queries the system for the number of bidders interested in the specific item. If there are any bidders, they are queried to determine the maximum bid \( b_{\text{max}} \). If \( b_{\text{max}} \) is greater than the seller’s sale price he/she initiates a transaction with the one of the bidders who offers \( b_{\text{max}} \). The first seller to request the completion of a transaction from a particular bidder can consummate the deal. The rest are re-
jected and must be reissued. If no bidder is found with a
bid high enough the seller uses a heuristic rule to determine
the new price for the current item, and then continues with
the next item available for sale. Since the rule requires the
minimum sale price for an item, the seller issues an ad-
tional query to retrieve the minimum sale price. Thus, un-
less a sale takes place, for each item a seller issues two
aggregate queries: one for maximum bid and one for mini-
imum asking price. The set of operations (queries/updates)
performed by a trader (bidder or seller) for a specific item is
called a session.

Bidders’ sessions are similar. For each item a bidder
queries the system to determine the number of sellers. If
there are sellers for this item, then the bidders query the
system for the minimum sale price and the maximum cur-
rent bid. After obtaining both values a heuristic rule is used
to determine the value of the bid. Then bidders wait for
their bid to be accepted by a seller of the specific item. If
not a new bid is determined. While waiting, bidders repeat
the cycle for their next item of interest.

Sellers replenish their supplies of items once they run
out. Similarly bidders pick new items every time they
manage to buy all the items they are interested in. In both
cases the traded items are chosen based on their popularity,
which is initially specified before the experiments start and
remains constant from then on. Thus, throughout the ex-
periment there exists a steady stream of trades.

The heuristic rule used by bidders and sellers is a sim-
ple rule with one learning element. For details refer to [21],
from which the rule’s parameters are taken. We preferred a
commonly used, and simple, agent over random increases
in bids and random decreases in sale prices for increased
realism. Additionally, an agent defines a reserve price that
is never exceeded.

5.2.2 Data Access
Traders export their bid or sale price with a query-able
XML document. Each time traders change their bid or sale
price they record it in this XML document. These docu-
ments also contain which items each trader is currently
trading. More specifically, we assume that each bidder
maintains the bid for a specific commodity under the path
\( p_{bid} = /\text{bidder/item/current} \_ \text{bid} \) and that each item contains
a unique identifier under the path \( p_{bid} \_ \text{id} = \\
/\text{bidder/item}@\text{item} \_ \text{id} \). Similarly, each seller also main-
tains an XML document in which item information is
stored. The sale price is stored in \( p_{sale} = \\
/\text{seller/item/sale} \_ \text{price} \) and there is also a unique id for each item in
\( p_{sale} \_ \text{id} = /\text{seller/item}@\text{item} \_ \text{id} \). Of course, each
trader’s document contains more information such as their
rating and the item description. Thus, traders form their
strategy by querying each other’s XML documents.

6. Experimental Evaluation
This section shows that the proposed aggregation frame-
work for P2P networks can significantly increase perfor-
ance and scalability in applications in which the M2M
query problem can arise. There are no claims in favor of the
economic efficiency of the proposed trading method. Simu-
lation results are presented first, while section 6.5 contains
experimental results from a prototype implementation of
the proposed framework.

6.1 Experimental Methodology
Our simulations employed a two-step methodology that
combines system measurements with simulation. n the first
step measurements were taken from a system that consists
of an XPath query engine, a catalog layer and an aggrega-
tion layer (Section 3). This system was loaded with both
trading data and catalog information. Then workloads of
XPath queries, catalog information lookups and aggregate
computations were executed and measurements are col-
lected, which yields the nominal peer performance. To ob-
tain variance across the peers during the simulations in the
P2P network the nominal performance is multiplied by a
factor \( sf \) that is uniformly distributed between 0.8 and 1.2.

The second step involved building a discrete event
simulation model using CSIM [5] consisting of nodes in-
terconnected with a DHT. The nodes in the P2P network
are modeled as single CPU, single disk workstations using
the measurements from the first step. The CPU service
discipline is round robin processing while the Disk is
modeled as a FCFS server.

6.2 System Measurements
The main goal of this step is to determine the relative per-
formance of catalog lookups and aggregate computations
relative to XPath queries. To implement this system we
used the Berkeley DB XML toolkit [1], which provides all
necessary data structures and XPath query functionality.
Therefore, we expect the relative performance difference to
be valid for various implementations.

The XMark benchmark data generator [27] was used to
generate data for auctions that have a structure similar to
our trading scenario data. The query load consists of XPath
queries that 1) retrieve the current bid for a specific item by
item ID, 2) retrieve seller and bidder information for an
ongoing trade by item ID or trader ID. A large number of
queries were generated by varying the identifiers of items
and traders. These queries model the XPath query load
that each node in the DCT scenario would be subjected to.
The size of the XML data used for measurements is 1 GB dis-
tributed across a large number of smaller files of about 0.5
MB each. Berkeley DB XML indices were created on item
and trader identifiers in order to speed up processing since,
without indices, the queries require more than 20 seconds
each. Thus, for each query the relevant document is re-
trieved using the index. Then, using the path the target
element is obtained.

<table>
<thead>
<tr>
<th>Table 2 Mean measured CPU and Disk service times</th>
</tr>
</thead>
<tbody>
<tr>
<td>Query Load</td>
</tr>
<tr>
<td>-----------</td>
</tr>
<tr>
<td>CPU</td>
</tr>
<tr>
<td>Disk</td>
</tr>
</tbody>
</table>

To measure catalog lookups and aggregate computa-
tions on a peer, the catalog layer was implemented using a
B-Tree in Berkeley DB as specified in [10]. The paths stored in the index are taken from schemas found in xml.org. Each stored path is associated with 1000 to 5000 random hypothetical node identifiers and each node contributes a random numeric value. These values were used to measure the cost of aggregate computations from data in the catalog index. The catalog lookup queries just retrieve the list of relevant nodes and the aggregate computations produce the node with the maximum or minimum value. As expected, aggregate computations are not measurably slower than plain catalog lookups, since the only additional work performed is the computation of the maximum or minimum from at most 5000 memory-resident values. The query load consists of random paths of various sizes chosen uniformly from the set of stored paths. The size of the catalog information index is 256 MB.

The system used for measurements was a 2.4 GHz Pentium 4 PC running Linux (kernel version 2.4.20) with an IDE Hard Disk and 512 MB of RAM. To model nodes in the P2P system, CPU and disk service times were computed as follows: 100 queries for both XML data and catalog information were executed and sorted by execution time. Then, significant outliers were eliminated. To obtain CPU mean service times the average of the middle 40% of execution times were used. To obtain disk mean service times the average of the top largest 40% of execution times were used. The results are presented in Table 2.

The mean service times obtained for CPU and disk requests are used to model updates and queries. The difference between queries and updates is that updates use the disk more frequently. Thus, XPath queries are disk bound 20% of the time while updates are disk bound 40% of the time. Similarly catalog queries are disk bound 5% of the time and catalog index updates 20%.

6.3 Simulation Setup

Simulations for two peer-to-peer versions and one central server system were implemented. This section describes their characteristics and their basic differences.

6.3.1 Peer-to-Peer Systems

Both peer-to-peer systems simulate a DHT that is a generic version of Chord [29]. Following the observation made in [15] we did not use a detailed network model, opting instead for a simple delay model where network delays are exponentially distributed with a mean of 50ms. We assumed that network bandwidth was not a limiting factor since only a small amount of data is transferred in each network message.

The impact of network volatility on a peer-to-peer system depends on the specific DHT implementation. In our case both systems are affected in the same way. Therefore our experiments examine stable peer-to-peer systems in order to obtain results that are independent of the underlying DHT implementation and demonstrate the raw impact of the Aggregation Layer framework in improving performance.

A catalog service as presented [10] is also present on both P2P systems and is used by traders to locate other traders. The catalog service maps XPath queries to DHT keys by using the identifiers of the traded items. The key difference of the two tested P2P systems is the absence of the aggregation layer in one of them.

The first P2P version that utilizes the aggregation layer has two variants: AL (Aggregation Layer) and LBAL. The difference is that the second variant employs load balancing (LB). Both use the catalog service to discover the traders for a given item. To retrieve the maximum bid and the minimum price the aggregation layer facilities are used. Thus, one catalog lookup is required to compute a query for the highest bid or minimum sale price. Load balancing is triggered using the criteria described in Section 4.3.1. The number of new keys created with each load balancing action is 20. Query load balancing using replication has been shown to be effective in [10] and is not tested here.

The P2P setup without an aggregation layer has variants GC (General Catalog) and GCI. GC utilizes the catalog service to discover traders, but directs XPath queries to the traders’ peers in order to collect data values and compute the maximum bid and the minimum selling price. These XPath queries are issued simultaneously to all traders and the aggregate is computed after the results are retrieved. This setup demonstrates the M2M query problem described previously. The GCI variant utilizes a local index that makes XPath queries for retrieving bids and sale prices as fast as the aggregate computations and catalog information lookups in the variants AL and LBAL. The index is used to obtain the value of frequently queried elements without needing to access the XML containers of the storage manager. In our case it contains the bids and sale prices indexed by item and trader identifier. The addition of a special index to heavily queried data is a common practice and also makes the comparison to AL and LBAL fairer, since our experiments indicate that XPath queries are about an order of magnitude slower than catalog lookups and aggregate computations (Section 6.2). Of course in the GCI variant both the fast index and the XML repository are updated during price updates, which yields a slightly higher average update time for GCI vs. GC. Load balancing is utilized with GC and GCI only for the catalog requests [10]. Load balancing XPath queries is impossible since the only way to answer these queries is to obtain data from the relevant data sources.

6.3.2 Central Server System

The central server system is intended as a reference point for evaluating the performance of the peer-to-peer variants. It consists of an ideal cluster of processors that are modeled exactly like the nodes in the peer-to-peer system (one CPU and one Disk). The cluster is assumed to be able to perform perfect load balancing of requests across its CPUs and Disks. The traders access their accounts from their workstations connected to the Internet, thus experiencing a network delay for each query and update request. The central server system also comes in two variants: CS and CSI. The CSI variant employs the fast local index described in Section 6.3.1. In the experiments that follow the central server sys-
tems had as many CPUs and disks as the corresponding peer-to-peer systems.

Graph 1 Seller session throughput for 10,000 nodes

6.4 Performance Results

The goals of this section are to determine the extent to which the aggregation layer improves performance and identify those cases where load balancing is required. A very important parameter in all configurations is the number of unique traded items $T$, which affects both P2P systems similarly. The smaller $T$ is, the larger, on average, is the number of traders for a particular item. The consequence for each node in GC and GCI, and for each aggregation point in AL and LBAL is more requests on average.

The first series of experiments involved peer-to-peer networks from 100 to 100,000 nodes. The numbers of bidders and sellers in the P2P network are approximately equal and each node hosts one trader (bidder or seller with equal probability). The number of items assigned to each trader is uniformly distributed between 5 and 15. The popularity of the traded items follows the 80/20 rule (a.k.a. Pareto’s principle) observed in many real world settings: 20% of the items are chosen by traders 80% of the time. The number of queued requests that trigger a load balancing action is $l_0 = 30$ and the number of new keys is 20. The centralized versions of the system have exactly the same trader and commodity distributions. The configuration with 10,000 nodes is presented first to demonstrate our key findings when varying the number of unique items.

The throughput of sessions for sellers (Graph 1) and bidders (Graph 2) as a function of the different number of items traded is the first set of results presented (relative error at most 1% with 95% confidence). The throughput for CS and CSI is insensitive to the number of unique traded items (dashed lines). For both P2P systems a small variety of items has an adverse effect as expected. Nevertheless, the impact of a small number of items is more significant on GC and GCI than on AL. For 1000 items, the throughput of AL is 9.6 times better than GCI while for 10,000 items AL is 1.5 times better. At the same time as the number of items decreases, the need for load balancing the overloaded aggregation point hosts becomes apparent: for over 2500 items LB and LBAL have similar performance. In the case of 100 items, however, LBAL is over 7 times better (Graph 2).

Graph 2 Bidder session throughput for 10,000 nodes

Table 3 shows the number load balancing actions. The 100 item configuration is especially demanding since it forces the system to create 8130 new keys in order to keep aggregation points from becoming overloaded.

Table 3 Load balancing actions (10,000 nodes, LBAL)

<table>
<thead>
<tr>
<th>#items</th>
<th>100</th>
<th>500</th>
<th>1000</th>
<th>2500</th>
<th>5000</th>
<th>10000</th>
</tr>
</thead>
<tbody>
<tr>
<td>#actions</td>
<td>8130</td>
<td>876</td>
<td>346</td>
<td>121</td>
<td>34</td>
<td>4</td>
</tr>
</tbody>
</table>

The following points can be concluded from the throughput numbers for 10,000 nodes:

- GC is virtually unusable. GCI is viable in the 5000 and 10,000 item setups but still lags far behind both AL and LBAL in terms of system throughput. The absence of data for GC (for 100 and 500 items) and GCI (for 100 items) was the result of event backlogs in the simulator, which led to high memory image sizes, forcing the simulations to abort. This indicates that the corresponding real system would not work.

- The benefits of LBAL are clearly visible for less than 2500 traded items. Above 2500 items the request rate at the aggregation points does not lead to the formation of a significant number of backlogs.

Table 4 Trader session durations for 10,000 nodes

<table>
<thead>
<tr>
<th>#items</th>
<th>CS</th>
<th>CSI</th>
<th>GC</th>
<th>GCI</th>
<th>AL</th>
<th>LBAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>3.13s</td>
<td>1.48s</td>
<td>N/A</td>
<td>N/A</td>
<td>15.60s</td>
<td>2.04s</td>
</tr>
<tr>
<td>500</td>
<td>47.1s</td>
<td>4.09s</td>
<td>N/A</td>
<td>4.97s</td>
<td>1.84s</td>
<td>N/A</td>
</tr>
<tr>
<td>1000</td>
<td>236.1s</td>
<td>229s</td>
<td>2.61s</td>
<td>1.79s</td>
<td>1.79s</td>
<td>1.79s</td>
</tr>
<tr>
<td>2500</td>
<td>153.4s</td>
<td>9.2s</td>
<td>1.73s</td>
<td>1.58s</td>
<td>1.58s</td>
<td>1.58s</td>
</tr>
<tr>
<td>5000</td>
<td>72.0s</td>
<td>4.41s</td>
<td>1.56s</td>
<td>1.52s</td>
<td>1.52s</td>
<td>1.52s</td>
</tr>
<tr>
<td>10,000</td>
<td>30.2s</td>
<td>2.24s</td>
<td>1.51s</td>
<td>1.49s</td>
<td>1.49s</td>
<td>1.49s</td>
</tr>
</tbody>
</table>

Load balancing has a much more significant impact on the throughput of bidder sessions than on that of seller sessions in the 100 node scenario (7 times versus 4 times). The reason is that bidder sessions do not include additional operations that occur during the finalization of a sale. These operations are part of the sellers’ sessions and cannot be load balanced like catalog lookups and aggregate computations.

Table 4 shows the average duration of traders’ sessions. Session durations are indicative of the usability of each configuration (Only sessions that do not contain sale finalizations are included. Sale finalizations take longer since they involve confirmation queries and additional updates). Short sessions imply short query response times, which in turn imply more accurate values for minimum sale price and maximum bid. The high session durations for GC and
GCI show that they are not usable. As expected, LBAL does a very good job of keeping the duration of sessions below 2.04 sec. Without load balancing, the average session duration for AL with 100 items climbs to 15.6 sec. Furthermore, Graph 3 shows the response times of catalog lookups and aggregate computations of AL and LBAL. The aggregation point hosts with 100 and 500 items become so overloaded that response times increase to 6.5 sec and 1.2 sec respectively. Thus, load balancing is not only beneficial for throughput but also for the accuracy of the aggregate computations since it manages to keep query response times low.

![Graph 3](image)

**Graph 3** Average response times for catalog and aggregation layer requests with and without load balancing

Note that in our experiments load balancing does not take into account the load of nodes when creating new keys. If it did, better results could be conceivably achieved in the demanding cases of 100 and 500 items, since highly loaded nodes would be avoided more effectively. Our experiments show that even randomly generated keys can help significantly. Load-aware key creation is left as future work.

![Graph 4](image)

**Graph 4** Speed-up of LBAL over AL for various network sizes and various numbers of items

While we have obtained results for a variety of network configurations, due to space limitations scalability results are summarized in Graph 4 and Graph 5. Graph 4 shows the speed-up of LBAL over AL in trader session throughput. The percentages in the legend denote the number of items in each network as a percentage of the total number of nodes. The graph, in essence, shows which combinations of network size and traded items make load balancing a necessity. For instance, in the 100 node network load balancing is not necessary. In the 500 node network the load balancing benefits are observable. In the larger networks load balancing of aggregate computations becomes a necessity as demonstrated by the achieved speed-up. Let $r = \text{number of unique traded items} / \text{number of nodes}$. In large networks (number of nodes > 1000) the smaller $r$ is, the larger is the speed-up of LBAL over AL due to load balancing. These results suggest that if $r \geq 25\%$ the speed-up achieved is not significant.

Graph 5 shows the trader session throughput speed-up of LBAL over GCI. The percentages have the same meaning as in Graph 4. The percentages are larger as GCI would not run on large networks (> 10,000) with a small variety of traded items. GCI appears somewhat usable when the number of traded items is large (> 50% of total number of nodes): With $r = 50\%$ GCI is about two to four times slower than the systems with the aggregation layer (AL, LBAL). For $r = 10\%$ or 25% GCI is clearly not scalable, which is indicated by the increasing speed-up.

![Graph 5](image)

**Graph 5** Speed-up of LBAL over GCI for various network sizes and various numbers of items

The usability of GCI improves as the number of traded items increases. Conceptually, if the variety of traded items becomes increasingly large, GCI would eventually perform similarly to AL since the number of aggregation point hosts would converge to the number of total nodes in the system, while at the same time the average requests per node would decrease in both cases. Graph 6 shows the throughput speed-up of LBAL over GCI for a 10,000 node network. The continued improvement of GCI with increasing item variety is confirmed. LBAL manages a combined trader session throughput of 11,700 sessions/sec and is so independent of the number of traded items. GCI starts out performing very poorly for 10,000 traded items: it is 11.2 times worse than LBAL compared to only 1.7 times in the case of a single user per peer (Graph 5). As the number of items increases GCI improves and finishes only 2.6 times slower than LBAL for 40,000 items.

![Graph 6](image)

**Graph 6** Speed up of LBAL over GCI for 10,000 nodes, 5 traders per node, and varying numbers of traded items

Graph 7 shows how the combined session throughput varies with the number of traders per node in GCI and AL. The network has 10,000 nodes and 50,000 unique items, which favors GCI. While AL starts out slightly better than GCI it becomes 4 times better (20 traders). Note, that while the throughput of GCI declines, the throughput of
AL increases slightly which means that AL can sustain additional users before the throughput peaks.

Graph 7 Combined trader throughput for 10,000 nodes, 50,000 unique items and varying number of traders

Graph 8 shows the combined trader session throughput rate for the 100,000 node network (one trader per node). The relative performance is similar to that of the 10,000 node networks. The CS and GC variants were omitted.

Graph 8 Combined trader throughput for 100,000 nodes and varying number of items

In conclusion, the performance of GCI demonstrates how badly the M2M problem can affect a system that otherwise has a powerful data source discovery mechanism and fast query processing on each of the peers. The large performance gain obtained using the AL and LBAL mechanisms shows that concentration points are necessary for applications that need summarized results from many data sources. The results show that DHTs can be effectively deployed to create and distribute computation points across large distributed systems.

Graph 9 Speed-up of PAL over PGC

6.5 Prototype Experiments

We implemented a prototype our system to confirm the simulation results. The system is written in Java and uses Pastry [24] as the DHT and Berkeley DB XML as the storage and query engine layer, which is accessed through the Java native interface. For the experiment we used 40 machines from our departmental Linux cluster similar to the one in Section 6.2. A trader with 15 commodities, on average, is emulated on each machine following the scenario outlined in Section 5.2. The non-aggregation layer configuration PGC corresponds to GCI in our simulations and the aggregation layer configuration PAL corresponds to AL. Due to space constraints we only present Graph 9 that shows the speedup of trader sessions of PAL over PGC achieved in the system with 40 nodes and a varying number of unique commodities. The aggregation layer achieves significant speed-up in a working prototype system and confirms the simulation results. Thus, our design can be used to enable novel Internet scale distributed application such as commodity trading.

7. Related Work

P2P pioneers such as Napster, FreeNet and Gnutella ([19], [8], [12]) have shown that loosely coupled peer-to-peer systems with thousands of nodes have significant potential for novel applications. Research followed with formulation of distributed location and routing protocols to address the initial shortcomings. These protocols ([23], [24], [29], [35]), known collectively as DHTs, guarantee a definite answer to a lookup query within a bounded number of network hops. Scalability and the absence of central infrastructure are their most cherished traits. DHTs can serve as solid foundation for scalable distributed applications.

The data management community has proposed the use of DHTs as a substrate for Internet scale distributed applications. The PIER project ([13] and [15]) advocates a DHT based query processor. Classic database system design requirements have been relaxed in order to achieve “organic scaling” to large numbers of nodes. Complementary to a P2P query processor are technologies for locating data sources in large peer-to-peer systems. A distributed catalog proposal based on multiple hierarchies can be found in [22]. Discovering peers’ data is performed using distributed catalog information based on multiple category hierarchies. Our previous work in [10] presents another approach for maintaining metadata information in peer-to-peer systems. This work builds upon DHT technology in order to distribute catalog information among nodes in a P2P system. Query load imbalance problems are identified and methods are proposed that take advantage of the structure of XML to load balance catalog queries. Our work extends on the load balancing mechanisms proposed in [10] by identifying situations in which updates can cause similar load imbalances and by proposing a load-balancing algorithm designed for updates.

Distributed file systems based on DHT protocols ([6], [18], [25]) first recognized the need for performing additional load balancing on top of the DHT layer to achieve scalability. Popular files are replicated across multiple nodes. We believe load balancing is inevitably a necessity in any distributed system that is designed with scalability in mind.

SCRIBE [26] is an application layer multicast publish/subscribe system that uses a DHT (PASTRY) to define rendezvous points for managing group communication on a
specific topic. It uses topic identifiers to assign topics to peers similarly to our use of catalog keys to assign aggregation points to peers. PeerCQ [11] uses DHT technology to assign continuous data monitoring queries to peers in a network. It formulates capability based matching of continuous queries to peers in order to achieve load balancing. Our approach is to perform load balancing as needed by reacting to high loads by distributing work to other peers. Finally, a basic difference between our approach and both SCRIBE and PeerCQ is that aggregation point hosts do not implement publish/subscribe functionality, and are thus much simpler. Their purpose is to passively collect data and maintain an always up-to-date aggregate.

Work in [17] presents distributed aggregate computations using gossip-based protocols in P2P networks. Its focus, however, is on how quickly aggregate computations converge to the actual value and not how to facilitate large volumes of aggregate queries over distributed data sets. Other studies of peer-to-peer systems ([4], [34]) use metadata on each peer to efficiently route searches to other peers or answer searches on behalf of other peers in the network. [3] defines data replication strategies to improve bandwidth requirements in ad-hoc peer-to-peer networks. In [9], [30] and [31] peer-to-peer architectures for efficient distributed search are presented. All proposed systems provide efficient and scalable designs for searching P2P networks using keywords or more complex containment queries and can benefit from the techniques presented here to address the M2M problem.

8. Conclusions

Looking at recent research results, it seems to the reality of highly scalable distributed systems with powerful mechanisms for locating and querying all available data is not far away. Our belief is that such systems will lead to exciting new applications that will in turn create new scalability bottlenecks. In this paper we presented the case for the many-to-many query problem that is bound to be a concern in very large distributed systems where queries require data from multiple data sources. Using existing technology we developed a framework that can solve this problem for a broad class of important queries by harnessing the resources of the peers in the distributed system. The experimental evaluation of our design on a distributed application shows that the M2M problem can degrade scalability and demonstrates the significant improvement in performance that can be obtained using the proposed framework.

References

[25] A. Rowstron, P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. SOSP 2001.
[32] XML Path Language (XPath) 2.0 http://www.w3.org/TR/xpath20/