

# Expressive and Efficient Peer-to-Peer Queries

Dennis Heimbigner  
Computer Science Department  
University of Colorado  
Boulder, CO 80309-0430, USA  
dennis.heimbigner@colorado.edu

## Abstract

*It is demonstrated how to provide a peer-to-peer system that supports an expressive query language while maintaining efficient distribution over a wide-area network. The key is to base message routing on the contents of the query messages and to use advertisements from resource providers to make that routing efficient. Advertisements are special queries that describe the data sets available at each site. Queries are encoded as messages that are efficiently distributed to sites providing advertisements. Distribution is determined by symbolically intersecting queries and advertisements. Performance measurements indicate that cost of symbolic intersection is low.*

## 1. Introduction

An important – perhaps even primary – application of peer-to-peer (P2P) systems has been to locate resources across a wide-area network. In the case of Gnutella [8] and Kazaa [14] these resources have often been MP3 music files, while Freenet [6] has been used as a more general distributed file system where the resources are specific files.

Simplifying somewhat, P2P systems operate by disseminating a *query* as a message to all the peer nodes participating in the P2P network. At each node, the query is evaluated against the local database of resources that are known to that peer. Each peer then responds to the query to indicate any resource matches. The concepts of query and response are quite general. A query is typically some form of declarative structure such as *bitrate*  $\geq 256$ . Responses can be returned in a variety of ways: by direct TCP connection to the query originator, by following the route by which the query arrived, or even disseminating the response using the same mechanism used for query dissemination.

The Achilles heel for P2P systems has been their inefficient distribution of messages [1][7]. The original Gnutella protocol was notorious for the load it put on networks because of its indiscriminate propagation of

messages. The inefficiency stems from the fact that with few exceptions, P2P systems do not exploit knowledge of the content of the messages being distributed. As a rule, P2P peers have no knowledge of the semantics of the messages, or in the cases where they could know, no advantage is taken of this knowledge. This requires them to distribute the query messages to every peer node because it is unknown which nodes could provide a response.

Our earlier work [11] demonstrated that knowledge of the message structure could be used to achieve more efficient distribution of messages, although at the cost of limiting the expressiveness of queries. The key contribution of this paper is to significantly extend the expressiveness of the queries that can be defined in our system without sacrificing its efficiency.

In that earlier work [11] we took advantage of *content-based routing* [3][5] (CBR), which is the basic technology underlying *publish-subscribe* systems. CBR is analogous to IP routing in the Internet, but instead of using IP addresses to determine the destination, the content of messages determines the destination (or destinations) for the message. As is discussed in Section 8 some forms of content-based routing have appeared in more recent P2P systems, but efforts in this direction are limited.

Using a CBR system as the underlying P2P infrastructure, we demonstrated that such a system could be used to mimic Gnutella, but with improved security, anonymity, and efficiency. This experience showed that CBR could provide a good substrate on which to implement query distribution. The one flaw in this hypothesis involved query expressiveness. Our initial effort only supported conjunctions of equality queries (e.g.,  $x=5 \wedge y=3$ ), which were useful, but we felt that further improvement was possible. This paper shows how the expressiveness was improved so that it is similar to the query languages used by other P2P systems, but more efficient.

The rest of this paper is organized as follows. First the general approach to using CBR for query distribution

is described and some preliminary details about the specific CBR, Siena, are provided. Next, an overview is provided about the operation of our query system. Given this background, the new query language is defined, and its implementation in our Siena CBR is described. Performance of the system is evaluated. Finally, there is a discussion of related work, and the current status of the system.

## 2. Query/Advertise: Content-based Routing for P2P

Our CBR infrastructure is based on the Siena system [2]. Originally designed as a publish/subscribe system, Siena has evolved into a more general content-based routing system. Siena provides a two-tier architecture similar to the super-peer models used increasingly in other P2P systems. In Siena, many peers (aka clients) are connected together via an overlay network of interconnected routers (aka servers). These routers are responsible for sending copies of messages to interested clients based on the message content.

The key to using CBR to efficiently distribute messages is to require each holder of resources to export some form of *advertisement* describing those resources and then disseminating those advertisements to other peers in the network. Advertisements look, syntactically, similar to queries. They implicitly describe the resources available at the site by describing an expression that evaluated to true when applied to that resource. They differ from client queries in their manner of use and distribution. Whereas queries move from the client to the resource holders, advertisements move from the resource holders to all the servers in the network. They are distinguished internally by markers in the protocol.

We can now see the outline of the operation of our CBR system when used as a P2P system to locate resources. Some clients (“queryers”) issue queries that are distributed to each advertising client whose advertisement (also a query) is deemed to “match” the queryer’s query. It is the job of the CBR system to ensure that queries are efficiently directed only to those data sources that *may* have information matching the query.

Upon receiving the query, the advertiser evaluates the query in two steps. First, the advertiser matches the incoming query against its advertisement query. If it is determined that no match exists at this step, then it is assumed that the advertiser has no data that will match the query and hence it can be ignored. The second step is to actually apply the incoming query against the actual datasets held by the advertiser. This might involve, for example, converting the query to an SQL query. It is important to note that the result of this query application may in fact be empty; this is acceptable and means that the advertisement was an approximate description of the

Message	Filter
{(artist, “Flatt and Scruggs”) (title, “Speedball”) (bitrate, 256) (MP3, true)}	{(artist,*, “Scruggs”) (bitrate, >=, 256)}

Figure 1. Example Message and Filter

data held by the advertiser. If the result of the query application is non-empty, then the advertiser responds to the queryer with that resulting dataset. As described elsewhere [11], the response may be returned through the publish/subscribe network but alternatively, it may be returned using some other mechanism such as a point-to-point TCP connection. The net effect is that the original query client receives, from multiple sources, zero or more datasets that matches its query. That client can then collate the responses to produce an aggregated result. This whole process involves a sequence of advertise-query-respond combinations, but we will refer to this simply as *Query/Advertise*.

### 2.1. Siena

Our Query/Advertise system explicitly builds upon the Siena content-based routing middleware system developed at the University of Colorado. Siena provides a convenient interface and offers important optimizations for improving the efficiency of message distribution. These optimizations are exploited to achieve specific efficiencies for query distribution.

Siena messages are structured as attribute-value pairs where attributes are simple names and the value is taken from a limited set of types. In standard Siena, the set of supported types is bool (true or false), long (64-bit integer), double (128-bit floating point), and byte-string, which also subsumes the more traditional string type. An example message could be represented as shown in the left column of Figure 1.

A client constructs a filter (a pattern) that specifies the kinds of messages it wishes to receive based on the message content. A filter is a set of triples of the form (attribute, operator, value). A filter matches a message if the value associated with each attribute in the message satisfies all corresponding filter triples that have the same attribute name. That is, for a given filter  $F$  and a given message  $M$ , the following holds.

$$\begin{aligned}
 &\forall \text{triples } (x, op, a) \in F & (1) \\
 &(\forall \text{pairs } (y, b) \in M \\
 &\quad (x = y \Rightarrow \text{Apply}(op, a, b) = \text{true})) \\
 &\text{where } \text{Apply}(op, a, b) = (a \text{ op } b)
 \end{aligned}$$

Table 1. Siena Filter Operators

Operator}	Argument Type
Equals (=)	bool, long, double, byte-string
Not-Equals (≠)	bool, long, double, byte-string
Less-Than (<)	long
Greater-Than (>)	long
Less-Equals (<=)	long
Greater-Equals (>=)	long
Prefix (>*)	byte-string
Suffix (*<)	byte-string
Contains (*)	byte-string
Any (any)	N.A.

The set of filter triples may be considered to be logically “and”ed together. A logical “or” can be achieved by specifying multiple separate filters. The right side of Figure 1 shows an example filter that would match the message on the left side. Table 1 shows the complete set of pre-defined operators available in standard Siena. Since they are used for matching, they all produce a boolean result.

It is important to note that the attribute names used in messages and filters have no inherent semantic meaning. As with all such attribute-based systems, there must be some external agreement about their meaning, and all parties must adhere to that agreement.

Siena adopts a two-tier peer architecture where arbitrary Siena routers connect to form a specific topology. In the simplest case, a client connects to a router and provides one or more filters. The router then forwards the filter to all of its peers. Each peer records where the filter came from, and forwards it to its peers. Later, when some other client connects to a router and generates a message, the local copy of the filter can be applied at that router to determine the next router to whom the message should be forwarded. If a message is generated for which no filter matches at the local router, then it will not be forwarded at all and so will generate no inter-router traffic. This kind of content-based routing is analogous to IP routing in the Internet, but instead of specific IP addresses, the content of messages determines the destination (or destinations) for the message.

### 3. Query Matching

Before discussing our particular chosen query language, it is necessary to understand how queries are used. The Query/Advertise system requires two interpretations of a query: *query application* and *query intersection*.

The first interpretation (query application) is the conventional one where a query is applied to a data set to produce a result set of data items matching that query.

The second interpretation, query intersection, is used to determine if a query “matches” (is relevant to) an advertisement. This will be used to determine the routing of a query within our Query/Advertise routers. Thus, given a query Q1 and advertisement Q2 (technically also a query) we say that Q1 intersects Q2 if the following holds.

$$\exists \text{datasource } d : ((Q1(d) \cap Q2(d)) \neq \emptyset) \quad (2)$$

That is, there exists a dataset, d, such that the result of applying Q1 to d and the result of applying Q2 to d have at least one data item in common. If Q2, say, represents the advertisement, then it makes sense to send Q1 (the query) to the advertising site because it may be able to provide a result.

In practice there are several things to note.

1. Determining query intersection using some actual dataset is impractical. Instead, the intersection is approximated by symbolically determining intersection based on the query expression and the advertisement expression. This means that there is no guarantee that the specific data set held at some site will actually satisfy Equation 1 even though the symbolic test indicates that the intersection should hold.
2. In order to avoid providing too many advertisements, a site may “fib” and provide an advertisement that technically covers more data than is available at the site. This allows an advertisement to further “approximate” its underlying dataset.
3. For more efficient matching, an advertiser may provide several advertisements such that the union of these advertisements represents the whole data set.

### 4. An Improved Query Language

The goal of this paper is to define a query language that is more expressive than our previous effort and that is as expressive as that typically used in other P2P systems. We also need to show that it can be efficiently implemented on a content-based router. To this end, it is necessary to review our previous query language [11].

The prior language co-opted the form of the

Advertisement	Equality Query
{(genre, =, “bluegrass”) (bitrate, >=, 256) (MP3, =, true)}	{(genre, “bluegrass”) (bitrate, 256) (artist, “Flatt and Scruggs”) (title, “Speedball”)}

Figure 2. Equality-based query and advertisement

Query
{(genre, (=, "bluegrass")) (bitrate, (>=, 256)) (artist, (*, "Scruggs"))}

Figure 3. Example query using improved query language

underlying Siena attribute-value pair messages. It took that form, but re-interpreted the semantics. Figure 2 shows an example of such an equality query and advertisement based on the same implicit music domain of Figure 1.

The message in the right column of Figure 2 represents a query of the following form.

*artist* = "Flatt and Scruggs"  $\wedge$  *title* = "Speedball"  
 $\wedge$  *bitrate* = 256  $\wedge$  *genre* = "bluegrass"

The advertisement in the left column of that same figure represents a purveyor of high quality bluegrass recordings in the MP3 format. The query can be intersected (Section 3) with that advertisement by evaluating the advertisement in the context of the values defined by the query message. In this case, the intersection is non-empty, so this query will match this advertisement. Note that any unmatched field (such as artist) is treated as intersecting.

The key is to note that such queries are restricted to conjunctions of equalities. Advertisements, on the other hand, can be more general and can involve conjunctions of triples using any of the operators in Table 1.

Our new query language is based on the filter language used in Siena; that is, we chose to introduce the triples used in filters as the basis for our query expressions. We did so because they are expressive, because they unify the query and advertisement languages, because they are easy to use for a user of standard Siena, and because they easily integrate into Siena while maintaining many of the desirable efficiencies provided by the Siena infrastructure.

Our specific approach was to introduce a *constraint* data type as a legal value for attribute-value pairs in a Siena message. A constraint has the form (*operator,value*), which is of course a filter triple without the attribute name.

Starting again with the right side of Figure 2, we can define a more interesting query against the same advertisement; this is shown in Figure 3. Now we can indicate a range for the bitrate and can indicate that we are interested in all recordings by "Scruggs". We will use the term *named constraint* to refer to such a pair whose value is a constraint.

In this model, query application is somewhat more complex than before because of the increased expressiveness of the query language. But the basic

process is still the same; the advertiser takes the query message and applies its named constraints to its data source to produce a response. The exact set of named constraints and the exact method of application are defined by the receiving site; it might convert it to an SQL query, for example, and apply it to a database of information about recordings.

The other interpretation of a query is for query intersection (Section 3). This determines if a given query message is applicable at given site as determined by the advertisements exported by the site. Recall our definition of a match between a filter and a message as defined in Equation 1. There we assume that each triple is matched against each pair with the same attribute name and a match is declared if all these individual matches succeed.

The new query language makes this process substantially more complex than with equality queries. However we can adapt our match procedure (Equation 1) to define query intersection for our new language. For a given advertisement A and a given query Q, the following revised equation holds if the query matches (intersects) the advertisement.

$$\forall \text{triples } (x, op1, a) \in A$$

$$(\forall \text{named constraints } (y, (op2, b)) \in Q$$

$$(x = y \Rightarrow \text{Intersects}(op1, a, op2, b) = \text{true}))$$

Note that we substituted the *Intersects* procedure for the *Apply* procedure in Equation 1.

So we say that a query and an advertisement intersect if each set of corresponding triples and named constraints intersect as defined by the *Intersects* procedure. This now reduces our task to defining *Intersects*(*op1,a,op2,b*) for every pair of operators (*op1* and *op2*) with arbitrary attribute values (*a* and *b*). We will defer further discussion of the *Intersects* procedure until Section 6 because we must develop some background first. As we shall see in the next section, we can co-opt the existing infrastructure of Siena to define *Intersects*.

## 5. Routing in Siena

We must digress slightly to discuss the details of routing in Siena so that we can later show how to efficiently route our new form of query.

The key to the routing process is matching each arriving message against a *selected* set of filters held at that router. As indicated in Equation 1 this involves invoking the *Apply* procedure on the filter and the incoming message. It turns out that Siena supports an optimization that allows it to avoid having to match the message against every filter in the system. This capability is embodied in a procedure called *Covers*. We explain each of these procedures – apply and covers – in the following two sections.

## 5.1. The Apply Procedure

Equation 1 requires the computation of expressions of the form  $Apply(op,a,b)$  for any operator supported in Siena (i.e., those in Table 1).

The *Apply* procedure for ordinary operators defines the ordinary application semantics of the operator. Thus, given two values  $a$  and  $b$  and an operator  $op$ , this procedure computes the value of  $(a \text{ op } b)$  (e.g.,  $(5 > 7)$ ).

During the match process, these values for  $a$  and  $b$  come from the filter and the message, respectively. In Figure 1, for example, matching the message and the filter involves computing several instances of *Apply*, including  $Apply(>=,256,256)$ , in order to check for a match over the *bitrate* attribute.

## 5.2. The Covers Procedure

The *Covers* procedure is required to support one of the forms of scalability provided by Siena. This procedure supports an optimization that can reduce the number of filters that a given router must maintain. Without this optimization, Siena would be forced to propagate all filters to all Siena routers.

The *Covers* relation between two filters F1 and F2 is the key to this optimization. The relation  $(F1 \text{ Covers } F2)$  holds if every message that matches F2 also matches F1. In other words, the set of messages matching F1 is a superset of the set of messages matching F2.

Since a filter is composed of triples of the form  $(x,op,a)$ , F1 covers F2 if the following holds.

1. Any attribute, A, that is in one filter but not the other is treated as if it was defined as always matching; that is it is treated as equivalent to  $(A,any,_)$ .
2. The set of values satisfying a triple from F2 is a subset of the set of values satisfying any similarly named triple from F1.

$$\forall \text{triples } (x,op1,a) \in F1$$

$$(\forall \text{triples } (x,op2,b) \in F2$$

$$(\forall z ((z,op2,b) = true \Rightarrow (z,op1,a) = true)))$$

We now define a *Covers* procedure with the following interpretation.

$$\begin{aligned} Covers(op1,a,op2,b) &= True \text{ if} \\ &(\forall z ((z,op2,b)=true \Rightarrow \\ &(z,op1,a)=true))) \end{aligned}$$

$$Covers(op1,a,op2,b) = false \text{ otherwise}$$

At a given router, the *Covers* relation forms a forest of partial order graphs over all the filters known at that router. Two filters F1 and F2 are in the same partial order graph if  $(F1 \text{ Covers } F2)$  or  $(F2 \text{ Covers } F1)$ . Otherwise, they are in different graphs in the forest. Siena routers need only propagate the most general filters, which are those that are at the root of each *Covers* graph.

In order to participate in the *Covers* optimization, each operator ( $op$ ) must define the procedure  $Covers(op1,a,op2,b)$  to compute if the *Covers* relation holds between two triples  $(x,op1,a)$  and  $(x,op2,b)$  from two different filters. This procedure assumes that (1) each triple has the same attribute name, and (2) that the operator in one or both of the triples is operator  $op$ .

It is important to note that the *Covers* procedure is optional, albeit highly desirable. Defining the *Covers* procedure to always return false is acceptable. The consequence, though, is that all filters containing that operator will be propagated to all Siena routers and significant inefficiencies may result. The *Covers* optimization has no downside because all the filters in the partial order graph represent actual filters; hence each filter has some client interested in messages matching that filter.

## 6. Implementing the New Query Language in Siena

In order to understand the implementation of our query language using Siena, it is first necessary to understand how queries and advertisements are mapped to the messages and filters of Siena. Specifically and critically, queries are mapped to messages and advertisements are mapped to filters. This means that we must do two things. First, we must correctly map our queries to the form of a Siena message (a set of attribute-value pairs) and each advertisement to a Siena filter (a set of attribute-operator-value triples). This requires extending the semantics of Siena messages and filters. Second, we must modify Siena to properly route our modified messages and filters.

The first step, then, in implementing queries in Siena is to introduce a new data type representing constraints. Recall that a constraint is a triple minus the attribute name:  $(>=,256)$ , for example. Adding this new constraint type is straightforward to implement and requires defining a new type in the type enumeration and defining serialization and de-serialization procedures for constraints.

The second step is figure out the effect of our new data type on the *Apply* and *Covers* procedures. Two questions arise in this context.

1. When we are matching a query (message) against an advertisement (filter), how do we know when to

Table 2. Intersect() Semantics (Partial)

Op1	Op2	Op1∩Op2	Rationale
=	op2	a op2 b	(only x = a works)
<	<	true	(any x < min(a,b) works)
>	>	true	(any x > max(a,b) works)
<	>	a > b	(any x ∈ range(a,b) works)
		...	

compute the normal *Apply* semantics and when to compute the *Intersects* semantics?

- How do we compute the *Covers* relationship between two advertisements?

Answering question 1 requires some kind of signal to indicate what to do, but we would like to do it in such a way as to minimize the disruption to the standard operation of Siena: we would like to be able to distribute queries and normal messages using the same set of Siena routers.

We use the presence of a constraint value in the message to treat the message as a query and thus as our signal to invoke intersection semantics. So assuming that the second argument comes from the message, we can define a revised *Apply* procedure as follows.

$$Apply(op1,(op2,a),b) = Intersects(op1,op2,a,b)$$

$$Apply(op,a,b) = (a op b)$$

The second equation is the standard interpretation used for all operators when a constraint is not involved. The first equation is used to invoke intersection semantics.

The remaining task with respect to *Apply* is to define *Intersects(op1,op2,a,b)*. We left this issue hanging at the end of Section 4, and now we have enough background to address it.

Recall that the idea is to try to find out if there is some value for  $x$  that can satisfy both  $(x op1 a)$  and  $(x op2 b)$ . Table 2 shows a subset of the *Intersects()* function; the values of  $a$  and  $b$  are assumed arbitrary. The first row, for example, says that  $(x,=,a)$  intersects  $(x,op2,b)$  is true if  $(a op2 b)$  is true; this is because the only possible value that can satisfy both is  $a$ . The second row says that  $(x,<,a)$  intersects  $(x,<,b)$  is always true because any  $x < \min(a,b)$  will satisfy both constraints.

Our last concern is to compute the *Covers* relation. In the Query/Advertise context, the *Covers* relation is being computed over advertisements and the question arises: is the *Covers* relation for filters directly applicable to *Covers* for advertisements? The short answer is yes.

To see this, we need to go back to the definition of an advertisement, which is that it describes a data set at a

source. Thus, we can say that for advertisements Ad1 and Ad2, *Ad1 Covers Ad2* if the data set described by Ad1 is a superset of the data set described by Ad2. Figure 4 illustrates this. If we propagate only Ad1 to other routers, then any query Q that intersects Ad2 will also intersect Ad1, so that the query will get directed correctly.

Since the *Covers* relationship purely computes a form of superset relationship, our existing *Covers* procedure can be used on advertisements to produce the correct result. The only difference is that for filters, the superset relationship refers to the space of messages and for advertisements it refers to the space of data sets.

## 7. Performance

It turns out that the externally visible performance of Query/Advertise and Siena [4] are dominated by the underlying network speed. Further, it turns out the extra processing for Query/Advertise at each node has no noticeable effect on performance with respect to Siena.

To see this last point, and in order to get some useful measurements, the *Intersects* procedure was extracted from the system and measured separately. That is, the measurements did not involve any network communication. The measurements purely involved the time to process a message through the Query/Advertise routing system.

Procedurally, the measurements were taken with the following features.

- Java implementation
- Millisecond wall clock resolution
- 1.5GHz Pentium M running Windows XP
- Randomly generated queries and advertisements.

To elaborate on the last point, each matched query and advertisement had N attributes, where N was the same for both the query and the advertisement. The reason for this is that extra attributes are quickly ignored in the *Intersects* procedure and so add no measurable cost. Each constraint associated with each attribute was constructed by randomly choosing a type, an operator and a comparison value. The constraints were different in the query and the advertisement although the types were the same.

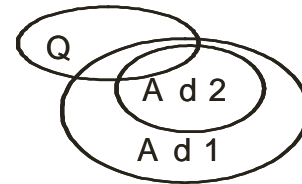


Figure 4. Advertisement covering

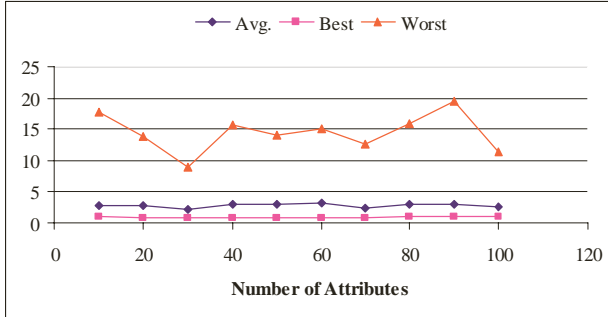


Figure 5. Total *Intersects* time ( $\mu$ sec): 10-100 attributes

Two measurements were computed for varying numbers of attributes. One measurement was the total comparison time and this is shown in Figure 6 where the number of attributes ranges from 10 to 100 in increments of 10. The other measurement was the time per attribute as shown in Figure 5. The latter figure was of course computed by dividing the per-message times in the first figure by the number of attributes in the message.

The clock resolution was in milliseconds; so in order to try to get microsecond resolution, each intersection was executed 100,000 times. In order to smooth the effects of randomly generated attributes, each complete run of 10-100 attributes was repeated 100 times and the average, best case, and worst case values computed. These are represented by the three lines in each figure; the top line is the worst case, the middle line is the average, and the bottom line is the best case. Repeated runs indicated that the time per attribute averaged under 0.1 microseconds. Similarly, the per-message time was about 3 microseconds for the larger messages. It is fair to say that that this cost is very small and so the cost for Query/Advertise is unlikely to be a problem.

## 8. Related Work

As indicated in Section 1, peer-to-peer systems often support the distribution of queries across the P2P network of peers. With few exceptions, these systems do not route based on message content and do not support advertisements.

The original Gnutella [8] is perhaps the most notoriously inefficient P2P system. It distributes messages indiscriminately, pays no attention to message contents, and makes no use of advertisements. Technically, Gnutella does not have a standard query language, but in practice, all of the implementations provide a query language that is similar to the new query language proposed in this paper. Given the similarity in expressiveness and the substantially greater efficiency obtained using Siena, it would appear that our system represents a significant improvement over Gnutella.

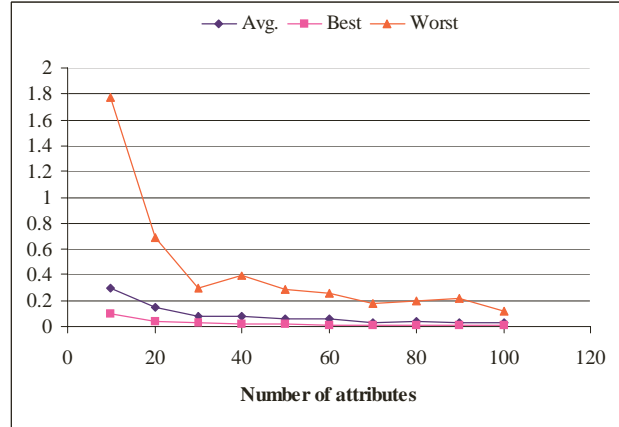


Figure 6. Per-attribute time ( $\mu$ sec): 10-100 attributes

The second version, Gnutella2 [9] is substantially better and is the basis for the Morpheus [15] P2P file sharing system. It does a certain amount of examination of messages but uses compressed hashing for routing. The values that are hashed are specific keywords identifying resources. Thus, it basically supports equality matching or inclusion matching (i.e. word X exists at site S). This is generally more efficient than our system for this class of query, but it cannot support the kinds of inequality searches our system provides.

Freenet [6] provides anonymous distributed file sharing. It is representative of a number of systems [18][19][22] based on distributed hash tables (DHTs). DHTs provide a much more restricted notion of query than any other peer-to-peer system: basically object equality tests. Clients ask for specific resources (identified by a unique hash) and the search process stops when that specific resource is found. Caching, load-balancing, and hop-routing can be supported to significantly improve performance. DHTs use message traffic about as efficiently as does Siena's content-based routing, and far more efficiently than Gnutella. It is apparently still an open question [1] if DHTs can be extended to support the more general queries provided by our new Query/Advertise query language.

Some efforts have been made to make P2P systems route based on message content. The DBGlobe [17] system, for example, utilizes multi-level Bloom filters at super-peers to control routing. This has the advantage that it can lead to very efficient routing. However, and not surprisingly, bloom filters represent a very restricted form of query based purely on set membership. As with Freenet, there is no obvious way to expand this to support inequalities.

At least one other P2P system [16] has focused on providing very expressive queries based on the W3C RDF standard. The problem here is that efficient routing becomes problematic because it would appear to require

solving the constraint satisfaction problem for two RDF queries, and as far as we are aware, this problem is not solvable except in very limited cases.

The Sun JXTA P2P system [10] provides an entirely different approach to query routing that theoretically allows for arbitrarily expressive queries while avoiding indiscriminate message distribution. The idea is that peers associate themselves with one or more *groups*. Each message that is sent out has an associated list of groups and it is delivered to the peers that are in any one of those groups. Internally, these group identifiers can be used to control routing. This is, of course, essentially equivalent to multi-cast routing where each JXTA group is mapped to an IP multi-cast address. This simulation of multicast, however, effectively limits the expressiveness of queries. The process for entering a JXTA network and for publishing a query to all the right groups is more complicated for the end-user than when using Query/Advertise or many of the other P2P systems. For a query domain of  $N$  attributes, JXTA would have to provide  $2^N$  groups to cover all the possible combinations of possible advertisements. Further, there would have to be front-end processing by the queryer to determine which groups it should associate with its query. This can all be made to work, but it is substantially more complex for the end-user than other P2P systems. The net result is a system that in practice forces users to use a query language that is more restricted than proposed in this paper.

## 9. Applications to other P2P Systems

Our Query/Advertise system is built upon Siena, but other publish/subscribe systems are available as alternatives upon which to build a Query/Advertise system. There are two issues here: scalability to wide-area networks (using some equivalent of the Covers relationship) and expressiveness. Most publish/subscribe systems are designed for local-area network use. Examples are Field [20] and ToolTalk [13]. Some other systems address are intended to operate over wide-area networks. Examples include TIBCO [23], Elvin [21], and Siena [2]. Both TIBCO and Elvin appear to suffer from a lack of automatic Covers relation support. The equivalent of the Covers relations must be manually established and maintained.

A more interesting question is: is this technique applicable to P2P systems that do not use publish/subscribe as the underlying paradigm: DHT-based systems or Gnutella, for example. There does not appear to be any way to do this without effectively re-inventing some variant of content-based routing that can route expressions over attributes. Gnutella, for example could be modified to do this, but there is no obvious feature of the Gnutella protocol that would provide any advantage.

## 10. Status and Future Work

A version of the Query/Advertise system described in this paper is available from the author as a modified version of Siena. Further improvements to the query language are being explored. These include limited support for disjunctive queries and adding variables to allow inter-triple value matching. As suggested by one of the reviewers, we are investigating the use of some variant of the constraint satisfaction language CLP(R) [12]. For our purposes, it would need to be extracted from its Prolog context and extended to handle string operations. It has some desirable properties; it is somewhat more expressive than our current language (at least for number domains) and yet appears to have linear time performance for intersection. This last point is important since anything past linear performance may be unusable in a P2P setting.

## 11. Conclusions

We have demonstrated how to provide a peer-to-peer system that supports an expressive query language, yet allows efficient message distribution across a wide-area network. The key is to base message routing on the contents of the query messages and to use advertisements from resource providers to make that routing efficient. Distribution is determined by symbolically intersecting queries and advertisements. We have implemented our system using the Siena content-based routing system. Performance measurements indicate that the cost for implementing query processing is low.

## 12. Acknowledgements

This material is based in part upon work sponsored by the Air Force Research Laboratories, SPAWAR, and the Defense Advanced Research Projects Agency under Contract Numbers F30602-00-2-0608 and N66001-00-8945. The content of the information does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

## 13. References

- [1] H. Balakrishnan, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Looking Up Data in P2P Systems. *CACM*, 46(2):43–48, 2002.
- [2] A. Carzaniga, D. Rosenblum, and A. Wolf. Achieving Expressiveness and Scalability in an Internet-Scale Event Notification Service. In *Proc. of the 19th ACM Symposium on Principles of Distributed Computing*, Portland OR., July 2000.
- [3] A. Carzaniga and A. L. Wolf. Content-based networking: A new communication infrastructure. In *NSF Workshop on an Infrastructure for Mobile and Wireless Systems*, number 2538 in *Lecture Notes in*

- Computer Science, pages 59–68, Scottsdale, Arizona, Oct. 2001. Springer-Verlag.
- [4] A. Carzaniga and A. L. Wolf. A Benchmark Suite for Distributed Publish/Subscribe Systems. Technical Report CU-CS-927-02, Department of Computer Science, University of Colorado, Apr. 2002.
- [5] A. Carzaniga and A. L. Wolf. Forwarding in a content-based network. In Proceedings of SIGCOMM 2003, pages 163–174, Karlsruhe, Germany, Aug. 2003.
- [6] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A Distributed Anonymous Information Storage and Retrieval System. In Proc. of the ICSI Workshop on Design Issues in Anonymity and Unobservability, Berkeley, CA, 2000. International Computer Science Institute.
- [7] P. Eugster, R. Guerraoui, A-M. Kermarrec, and L. Massoulié. Epidemic Information Dissemination in Distributed Systems. *IEEE Computer* 17(5): 60-67 (May 2004).
- [8] Gnutella Home Web Page. <http://gnutella.wego.com/>.
- [9] Gnutella2 Developers Network. <http://gnutella2.com>.
- [10] L. Gong. JXTA: A Network Programming Environment. *IEEE Internet Computing*, 5(3):88–95, 2001.
- [11] D. Heimbigner. Adapting Publish/Subscribe Middleware to Achieve Gnutella-like Functionality. In Proc. of the 2001 ACM Symposium on Applied Computing (SAC 2001), Las Vegas, Nevada, 11-14 March 2001.
- [12] J. Jaffar, S. Michaylov, P. Stuckey, and R. Yap. The CLP(R) Language and System. *ACM TOPLAS*, 14(3):339-395 (July 1992).
- [13] A. M. Julienne and B. Holtz. ToolTalk and Open Protocols, Inter-Application Communication. Prentice-Hall, 1994.
- [14] Kazaa Home Web Page. <http://kazaa.com/>.
- [15] Morpheus Home Web Page. <http://www.morpheus.com/>.
- [16] W. Nejdl, W. Siverski, and M. Simtek. Design Issues and Challenges for RDF- and Schema-Based Peer-to-Peer Systems. *SIGMOD Record*, 32(3):41–46, 2003.
- [17] E. Pitoura, S. Abiteboul, D. Pfoser, G. Samuras, and M. Vazirgiannis. DBGlobe: A Service-Oriented P2P System for Global Computing. *SIGMOD Record*, 32(3):77–82, 2003.
- [18] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content Addressable Network. Proceedings of the SIGCOMM '01 Symposium on Communications Architectures and Protocols, Aug. 2001, San Diego, CA.
- [19] A. Rowstrom and P. Druschel. Pastry: Scalable, Distributed Object Location and Routing for Large Scale Peer-To-Peer Systems. *ACM International Conference on Distributed Systems Platforms*, Nov. 2001, pp. 329–350.
- [20] S. P. Reiss. Connecting Tools Using Message Passing in the Field Environment. *IEEE Software*, pages 57–67, July 1990.
- [21] W. Segall and D. Arnold. Elvin Has Left the Building: A Publish/Subscribe Notification Service with Quenching. In Proceedings of the 1997 Australian UNIX Users Group, Brisbane, Australia, Sept. 1997.
- [22] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications. Proceedings of the SIGCOMM '01 Symposium on Communications Architectures and Protocols, Aug. 2001, San Diego, CA.
- [23] TIBCO, Inc. Rendezvous Information Bus, 1996. <http://www.rv.tibco.com/rvwhitepaper.html>