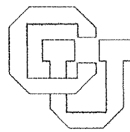
Adapting Publish/Subscribe Middleware To Achieve Gnutella-like Functionality

Dennis Heimbigner

CU-CS-909-00



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

ANY OPINIONS, FINDINGS, AND CONCLUSIONS OR RECOMMENDATIONS EXPRESSED IN THIS PUBLICATION ARE THOSE OF THE AUTHOR(S) AND DO NOT NECESSARILY REFLECT THE VIEWS OF THE AGENCIES NAMED IN THE ACKNOWLEDGMENTS SECTION.

Adapting Publish/Subscribe Middleware to Achieve Gnutella-like Functionality

Dennis Heimbigner
(dennis@cs.colorado.edu)

CU-CS-909-00

1 September 00

University of Colorado at Boulder

Technical Report CU-CS-909-00
Department of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309

Abstract

Gnutella represents a new wave of peer-to-peer applications providing distributed discovery and sharing of resources across the Internet. Gnutella is distinguished by its support for anonymity and by its decentralized architecture. The current Gnutella architecture and protocol have numerous flaws with respect to efficiency, anonymity, and vulnerability to malicious actions. An alternative design is described that provides Gnutella-like functionality but removes or mitigates many of Gnutella's flaws. This design, referred to as Query/Advertise (Q/A) is based upon a scalable publish/subscribe middleware system called Sienab. A prototype implementation of Q/A is described. The relative benefits of this approach are discussed, and a number of open research problems are identified with respect to Q/A systems.

1 Introduction

Gnutella [9] represents a new wave of *peer-to-peer* [19, 16] applications providing distributed discovery and sharing of resources in wide-area networks such as the Internet. Gnutella is distinguished by its support for anonymity and by its decentralized architecture. This is in contrast to Napster [18], for example, which is similar, but is more centralized. The Gnutella protocol provides a simple request-response paradigm for sharing files directly between peer computers. Users send out requests for files; these requests are propagated to all peer nodes in the Gnutella net, and in response, nodes generate replies back to the query originator indicating that they have the specified file or files.

Gnutella is biased towards the sharing of files, but it and similar applications are being used for managing other kinds of resources. Intel [14] for example, is using the technology to solve chip design problems by finding and using spare cycles on the company's machines. A Peer-to-Peer Working Group [19] has been formed to standardize and exploit this kind of application, with Hewlett-Packard, IBM, and Intel as founding members [16].

The promise of Gnutella is marred by a large number of flaws in its protocol and architecture. In particular, the following problems have been identified [10, 11].

- *Imperfect anonymity.* In practice, the anonymity of Gnutella is easy to compromise because node IP addresses are revealed at various points in its operation.
- *Malicious users.* Since every node potentially routes messages from other nodes, it is easy for malicious users to seriously affect the core operation of the Gnutella net.
- *Efficiency.* Gnutella has a well-deserved reputation for generating inordinate amounts of network traffic. In addition, slow nodes can create serious performance bottlenecks in the net.
- *Query expressions.* The format of Gnutella queries is not standardized. A query contains an arbitrary character string whose interpretation is entirely determined by any node which receives it.

The goal of this paper is to demonstrate that these flaws in Gnutella can be alleviated using a scalable *Publish/Subscribe* [4] middleware system which can be adapted for distributed routing of messages across a wide-area network such as the Internet. The Publish/Subscribe paradigm is implemented by such systems as Siena [2], Tibco [22], and Elvin [21].

In a publish/subscribe system, clients publish *event* (or *notification*) messages with highly structured content, and other clients make available a filter (a kind of pattern) specifying the *subscription*: the content of events to be received at that client. Event message distribution is handled by an underlying *content-based routing* [6] network, which is a set of server nodes interconnected into a peer-to-peer network. The content-based router is responsible for sending copies of event messages to all clients whose filters match that message.

Analogous to publish/subscribe, the *Query/Advertise* (Q/A) paradigm is defined to represent Gnutella-like systems. *Queries* are represented by the contents of messages describing attributes of resources (e.g., files) of interest. Providers of resources establish *advertisements* describing the attributes of their available resources. Upon receiving a query, providers generate *respons* messages whose content describes the detailed attributes of their specific resources.

These responses are injected into the network and routed back to the query originator. The provision for responses represents a major difference between Q/A and publish/subscribe.

This paper is organized as follows. First, the architecture, operation, and problems of Gnutella are described in more detail. Then the architecture and operation of a specific publish/subscribe system is described. Next, it is shown how to implement Query/Advertise on a such a system and how it solves many of Gnutella's problems. A prototype implementation is also described. After that, research issues for Q/A are discussed, and finally there are some comparisons to related systems.

2 Gnutella Architecture and Protocol

The Gnutella architecture consists of a dynamically changing set of nodes connected using TCP/IP. Each node acts as a client (an originator of querys), a server (a provider of file information, and as a router (a transmitter of queries and responses). The generic term *node*[1] will be used to refer to this combination of functionality. At any given point in time, a Gnutella net consists of a set of interconnected nodes. A new Gnutella user starts an instance of the Gnutella node software. That node uses out-of-band means to locate another node and establish a connection to it. This extends the net and makes the new node's files available to all other nodes in the net.

Once connections are established, nodes use the Gnutella protocol to communicate. There is an initialization conversation following which nodes send out typed packets into the Gnutella net to locate and retrieve files. There are five kinds of packets.

1. QUERY – request to locate a set of files matching some filter criteria.
2. HITS – response to a query giving a list of files matching the filter criteria and the IP address of the provider; note that there may multiple responders to a given query.
3. PING – request the transitive closure of connected nodes to identify themselves.
4. PONG – response by a node upon receiving a PING; the responding node provide its IP address and number of sharable files it contains.
5. PUSH – request a file provider to contact the requester. This provides a simple mechanism to attempt to get through firewalls.

The PING, PONG, and PUSH packets are not considered further as they are not essential to the core operation of Gnutella.

The query-response cycle for a Gnutella node involves 3 steps. In the first step, a QUERY packet is sent out. The packet contains a string specifying the set of files of interest. Each node that receives the packet uses this string to determine which files, if any, match the query. Unfortunately, Gnutella defines no standard format or matching semantics for this string; its interpretation is completely determined by each node that receives it. In practice, the query string is interpreted as a literal substring or as a regular expression that is to be matched against local file name paths.

[1] Some developers of Gnutella use the term *Servent*.

In the second step, a node that matches a query generates one or more HITS packets giving information about obtaining specific files. Again, there is no standard format, except that it is a list of null-terminated strings. As a rule, these strings specify URLs for each file, but because URLs contain IP addresses, the HITS packet breaks the anonymity of nodes.

In the third step, the query node connects directly to the response node and uses a simplified version of the HTTP [8] protocol to retrieve the file using the returned URL. Thus, it bypasses the Gnutella net altogether.

Message passing in Gnutella represents a form of spreading activation. That is, whenever a node receives a message, it sends copies out to all of its other connections. Obviously this can generate large amounts of redundant traffic. Gnutella relies on two mechanisms to reduce traffic. First, each message has a time-to-live (TTL) counter that is decremented on every transmission. Secondly, Nodes are expected to cache information about messages they receive and if they receive a duplicate, then they do not forward it. In spite of these features, Gnutella still generates a large number of messages. The protocol expects that response messages, such as PONG and HITS will not be widely disseminated, but will instead by returned along a specific path back to the generator of the original PING or QUERY message. This requires, of course, that intermediate sites track path information so that this kind of directed reply can be implemented. It has been observed that many available Gnutella software packages do not keep such information, thus causing reply messages to be widely propagated, which again increases message traffic unnecessarily.

3 A Publish/Subscribe Architecture

The Siena [2] publish/subscribe middleware system ,developed at the University of Colorado, will be used to provide a canonical architecture. Other similar systems exist and are discussed in Section 8. We provide a somewhat detailed description of Siena in order to present the features necessary for its use in Query/Advertise.

Siena messages are structured as attribute-value pairs where attributes can have one of many possible types, such as string, date, integer, or double. An example message could be represented as the following set of tuples.

{ (author,"John Steinbeck") (title,"Grapes of Wrath") (firstedition,True) }

A client establishes a subscription by specifying a filter pattern that specifies the kinds of messages it wishes to receive. A filter is a set of triples of (attribute, operator, value), where the operator is a comparison operator such as the usual arithmetic comparisons for numbers, or *substring* for strings. In order for a message to match a filter, every attribute in the message must satisfy all corresponding filter triples when the message value is substituted and the operator applied. Thus, all of the tuples may be considered to be logically ANDed together. A logical OR can be achieved by specifying multiple separate filters. An example filter might be represented by the following set of tuples.

{ (author,=,"John Steinbeck") (firstedition,=,True) (price,>,$400.00) }

It is important to note that the attribute names used in events and subscriptions 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 peer-to-peer architecture where arbitrary Siena servers connect to form a specific topology. In the simplest case, a client connects to a server and establishes a subscription. The server then forwards the subscription filter to all of its peers. Each peer notes where the subscription came from, and forwards it to its peers. Later, when some other client connects to a server and generates an event message, the local copy of the filter can be applied at that server to determine the next server to which the message should be forwarded. Note that if a message is generated for which no filter matches at the local server, then it will not be forwarded at all and so will generate no inter-server 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 of determines the destination (or destinations) for the message – an important distinction with respect to anonymity.

Siena is specifically designed to scale well to wide-area networks. One way this is achieved is by providing an important optimization that can reduce the number of filters that a given server must maintain. Key to this optimization is the *Covers* relation over filters. At a given server, for any two filters, F1 and F2, say, it can be determined if F1 *Covers* F2 or F2 Covers F1, or neither. F1 Covers F2 if any message that matches F2 also matches F1; F1 is more general than F2. Using this relationship, a forest of partial order trees can be constructed over all filters. Siena servers need only propagate the filters that are at the root of each *Covers* ordering.

4 Query/Advertise using Publish/Subscribe

A query/advertise system has three primary concepts: advertisements, queries, and responses. Implementing Query/Advertise (Q/A) on top of a publish/subscribe messaging system involves mapping these three concepts to the concepts of the underlying publish/subscribe system.

1. *Advertisements.* Advertisements map to subscriptions. A client acting as a resource provider describes his available resources using a filter pattern and establishes a subscription based on that filter. In effect, the notion of a subscription is generalized to become a kind of *content-based address* where messages are sent to addresses based on their content. For Q/A, this address represents an advertisement describing queries for which it may be able to provide a response.

2. *Queries.* Queries map to event messages. A client constructs a message describing the resource in which it is interested, and then inserts the message into the Q/A system where it is distributed to all clients with matching advertisements. As with advertisements, the notion of event is being generalized to a message and is not tied specifically to event semantics. Note that the query must contain some kind of identification, a "return address" if you will, so that responses can be returned to the query originator.

3. *Responses.* Responses also map to (event) messages in the underlying publish/subscribe system. Each advertiser that receives a message must perform a detailed examination of

its stock of resources and construct a message describing each available matching resources in more detail. This response is then routed back to the query originator. The response must include some identification for the matching advertiser so that the query originator can obtain the actual resource. For both query return addresses and advertiser addresses, note that the identity is generally not an IP address, but rather some arbitrary but unique set of attributes sufficient for routing, but capable of maintaining anonymity.

The above discussion glosses over some mismatches between a publish/subscribe system and a Q/A system. That discussion is deferred to Section 7.

5 A Prototype Implementation

A prototype Q/A system called *Quad* was developed using the existing publically available Java-based Siena prototype [3]. Much of the underlying Siena architecture and design shows through in Quad. This prototype supports query, advertise, and response, but does not support actually copying a resource such as a file from a responder to a query originator.

Most of the details of the implementation are relegated to the appendices of this paper, and even then, some inessential details are elided. Appendix A show the basic Quad application programming interface (API) as a Java package called *quad*. The class *Quad* is the primary class, and defines the basic interface for both query clients and advertising clients. Following class Quad, a number of auxiliary classes define Advertisements, Messages and Addresses. Messages serve to define both queries and responses to queries. Addresses serve to define the query originator so that responses can be correctly routed.

Appendix B shows the essential code for an Advertising client. As with all Java programs, the initial entry point is the static procedure named main. This procedure obtains two property values from the environment: a special kind of URL pointing to the local Q/A server, and an identifying string for the client. Continuing in the main procedure, an instance of class Quad is created and an instance of an Address is created to serve as additional identification for the Advertiser. Next, an advertisement is created and the advertisement is given to Quad.

At this point, the advertising client enters a loop and blocks waiting for a query to appear. The query is obtained and matched against a set of instances of a class called Book (not shown). For each matching book, a reply is generated describing the details of the book. The return address of the query originator is implicitly copied in the "quad.reply()" call from the query message to the reply message. This allows the Q/A servers to properly and efficiently route the reply back to the query originator. Similarly, the identity of the advertiser is also copied from the advertiser's Address object into the reply. This allows the query originator to communicate with the specific provider to obtain access to the resource. Of course, the provider could mimic Gnutella and provide a direct URL to the resource, but that would seriously compromise anonymity.

Appendix C shows the essential code for a Query client. The initial part is similar to the Advertising client. The Address object serves as the destination for replies to a query. A query message is created and marked with this return address, and then the message is sent to the local Q/A server.

For this prototype implementation, query clients invoke a "wait()" function to block for a specified number of seconds. This allows time for replies to be returned from matching advertisers. Alternatives to waiting can be designed, and this issue will be discussed in more detail in Section 7. Finally, the advertising client loops and extracts replys and prints them.

Appendix D shows sample output from executing the query client and advertising client examples, respectively. It is assumed that the advertiser is started first, and then the query client. Prior to starting either, it is assumed that one or more Q/A servers have been started. Since this implementation does not require modifying the server implementation, it is sufficient to create a standalone server using the existing Siena *StartServer* mechanism. For more details, refer to the Siena documentation [3].

6 Comparing Q/A to Gnutella

The primary operational distinction between Gnutella and Q/A is the use of explicit advertisements. Gnutella file providers give no previews about their available files. This forces all query messages to be delivered to all providers and the equivalent of filtering is only performed at the last step by the provider.

The management of anonymity in Q/A differs from Gnutella in an important way. As indicated in Section 2, Gnutella's anonymity is limited since messages use explicit IP addresses, causing them to be visible to all clients. In Q/A, through the underlying publish/subscribe system, clients only need to provide their IP address to the first level server to which they connect. After that, all references to the client are in terms of content-based advertisements and addresses. This effectively hides the identify of clients from all other servers and clients comprising the network.

The primary architectural distinction between Gnutella and Q/A is the separation of clients and servers. Q/A assumes a set of distinguished nodes that are running a specific server software package. The servers are interconnected and use a protocol that is different from the one used between client and server. Q/A clients use some out-of-band mechanism to locate a server. The client connects to that server and generate messages and provide advertisements for receiving messages. It is important to note that Q/A clients do not participate in message routing. That is functionality reserved to the servers. The architecture is further distinguished by its ability to selectively route messages based on forwarding of advertisements and on optimizations provided by the Covers relations among advertisements.

We can now revisit our list of Gnutella flaws and see how Q/A solves many of these problems.

- *Anonymity.* Q/A's anonymity is more complete than is Gnutella's because only a single server needs to know the IP address of a client.
- *Efficiency.* The use of advertisements and the Covers relation significantly reduces the traffic in a Q/A net as opposed to a Gnutella net. Q/A also automatically routes replies along the path from the advertiser to the query originator.
- *Malicious Users.* If the set of servers is controlled carefully, the impact of a malicious client can be limited because it can only affect the first level client-server communication. Malformed messages from a client can be caught and suppressed by its server. Also, any

attempt by a client to flood the network with queries can be metered by a server, and queries for which there is no possible provider will be suppressed and not propagated since no filter will match.

- *Query Expressions.* Q/A query expressions are standardized by the underlying publish/subscribe system so there is never any misunderstanding about their interpretation. There is, however an issue about their general expressiveness (see Section 7.2).

The Q/A approach has many advantages and fixes many of the problems of Gnutella-like systems, In fairness, however, there are some advantages for the Gnutella approach that a Q/A system would currently find difficult to emulate.

One advantage is the ability of clients to also act as servers and to dynamically extend the Gnutella network. Q/A inherently requires a separation of servers and clients, and it is this separation that provides many of the Q/A advantages. Still, this is a very desirable property and worthy of further study.

A second advantage concerns caching. Some variants of the Gnutella architecture support caching of files at more than one client. This is possible because of the spreading activation model, because clients are part of the routing infrastructure, and because many queries are requests for specific files. Adding caching to the Q/A architecture is potentially possible, but would appear to require major changes in the current message processing and the embedding of knowledge about resources into the Q/A servers. In effect, a mobile resource would need to carry its advertisement along with it.

7 Research Issues

Although this paper presents a reasonable design and implementation for Q/A using a specific publish/subscribe system, there is clearly much room for additional research and for alternative designs with other desirable properties. This section identifies some of those research issues and discusses some possible alternatives.

7.1 Response Collection

Replys to queries are returned asynchronously with respect to the original query. So an important problem is determining when all responses to a query have been received. Depending on client speed, client response size, server routing, and network size, responses can appear arbitrarily long after the original query.

We can identify a number of standard solutions that help address this problem of collecting responses.

- *Timeout* – wait for a specified period of time and take whatever responses have been received by that time.
- *Number-of-Responses* – wait until a specified number of responses have been returned. This assumes that the network is reliable and that all – or enough – clients are up to provide the required number of responses.
- *Two-Phase-Query* – wait for a period of time and then send out a follow-up query that forces responders to generate an immediate response.

- *Quick- Reply* – upon receiving a message, advertisers generate an immediate reply saying, in effect, "I will reply."
- *Extended-Query* – accept arbitrarily late responses. This presumes that it makes sense to process query responses individually.

In many cases, some solutions can profit from or require the use of other solutions. Two-phase-query, for example, requires the use of timeout, and number-of-responses could profit from quick-reply information.

7.2 Query, Advertisement, and Response Expressiveness

Using simple attribute-value pairs for querys may not be as expressive as required for realistic queries. Instead, one might prefer to have queries that can specify such things as regular expressions for strings or value ranges for numbers. On the advertisement side, tuples with comparison operators are already supported, but again expressiveness may be lacking. The problem with moving to more expressive queries and advertisements is that it can significantly complicate routing by the Q/A servers because it makes computing the Covers relation difficult and reduces opportunities for optimization.

In the event that an advertiser has multiple matching resources, it must return multiple responses to the query originator. The simplest approach is to generate multiple responses: one for each matching resource. This is the approach taken in the prototype (Section 5). The problem with this approach is that it leads to significantly more message traffic. An alternative is to encode all of the responses from a given advertiser into a single reply message. Given the simple attribute-value pair format of our prototype, this encoding is non-trivial. Solving this requires adding some form of list or vector type to the current format.

7.3 Resource Retrieval

Even after a client has received responses to its query, it still has the task of retrieving that resource from some responding client. Conceptually, the simplest solution is for the provider of the resource to send its contents back through the Q/A network to the consumer of that resource. This assumes, of course, that such transmission makes sense for the resource. Using the Q/A network maintains complete anonymity and uses the same infrastructure for all communications, but at the cost of increased message traffic.

In Gnutella, the retrieval process occurs out-of-band. That is, the query node connects directly to the response node and uses a simplified version of the HTTP protocol to retrieve the file. Q/A could use such an approach, but naively implemented, it significantly reduces client anonymity because one or the other of the clients must reveal its IP address so the connection can be made. One possible solution is to use a well-known (and trusted) third party (e.g., one of the Q/A servers). Both clients connect to the third party, which then provides a channel for passing information from one client to the other. This has the advantage of keeping the initial clients anonymous with respect to everyone except the third party.

7.4 Malicious Activities

Although Q/A can prevent many malicious activities, it shares with other Gnutella-like systems some vulnerabilities that are harder to stop.

- *False Advertising.* Consider, for example, a client that advertises for the equivalent of "all messages." When it receives a message, it responds with a bogus reply containing an advertisement for a pornographic site. A Q/A server could mitigate this by requiring all advertisements to have a degree of specificity. For popular resources, this might not provide much protection.
- *Freeloading.* It has has been found empirically [1] that most users of Gnutella retrieve resources, but do not themselves ever provide any resources to others. A market-based approach, such as used by MojoNation [17, 15], might alleviate this problem, but adding this kind of capability to a Q/A or the routing infrastructure of a publish/subscribe system remains an open research issue.

7.5 Performance

The performance of the simple Siena-based prototype is adequate for small scale use. Some Siena simulations [5] exist that provide plausible evidence that the underlying content-based routing network is scalable. Similar simulations combined with performance measurements are needed to verify the scalability of a Q/A system.

7.6 Long-Term Queries

The current Q/A model as described in this paper requires a query client to periodically send out a query in order to detect new replies that may have become available since the last query. It would be more desirable if this could be automated in some fashion so that new replies could be sent as soon as they become available.

One way to achieve this is to combine Q/A directly with the Publish/Subscribe paradigm. Thus, some queries could be marked as, say, *persistent*, so that they would remain in force and would continue to receive replies until rescinded. Operationally, this would be implemented by turning the query into a subscription, and introducing a mechanism by which advertising clients could inform their interface of new resources. If the new resource matched a persistent query, then an event message (in the publish/subscribe sense) would automatically be generated and sent to the query client.

8 Related Work

The closest systems related to Gnutella and Q/A are Napster [18, 24] and Freenet [7]. Unfortunately, little information about these systems exists outside of the World Wide Web. So for those web-based references, the bibliographic citation includes a specific URL.

The primary difference between Napster and Gnutella is in the architecture. Napster uses a restricted set of centralized servers to do the query and advertise matching. It is similar to Gnutella in that the files are held in user established clients and the set of clients can grow and shrink dynamically.

Freenet provides anonymous distributed file sharing, and it is extensible by user-established clients. It has a much more restricted notion of query than does Q/A: clients ask for specific files (identified by a unique hash) and the search process stops when that specific file is found. Caching is also supported. Freenet uses message traffic about as efficiently as does Siena's

content-based routing, and far more efficiently than Gnutella. It is not clear how freenet could be extended to support more general resource discovery.

MojoNation [17, 15] is a more recent Gnutella-like system that attempts to reduce message traffic and freeloading by adding a charging model. Downloading a file incurs "costs" and providing a file for others provides "money" to use against costs.

The Quad Q/A system is built upon Siena, but other publish/subscribe systems are available as alternatives upon which to build a Q/A system. There are two issues here: scalability to wide-area networks and expressiveness. Most publish/subscribe systems are designed for local-area network use. Examples are Field [20] and ToolTalk [13]. Two systems other than Siena address the wide-area network issue: TIBCO [22] and Elvin [21]. The primary problem with both is their lack of automatic Covers relation support. The equivalent of Covers relations must be manually established and maintained.

Expressiveness is a problem for *subject-based* publish/subscribe systems. These systems, of which TIBCO is an example, provide only a single content string (the subject) for use in routing. This severely limits expressiveness, and it is not clear if any sort of reasonable Q/A system could be built using a subject-based system.

Resource discovery systems are tangentially related to Q/A. Jini^{tm} [12, 23] is perhaps the best known of these systems. Jini defines a collection of programming interfaces. The implementations behind them are prototypes that do not appear to have addressed issues such as wide-area scale and message traffic. In principle, there is no reason that the resource discovery part of Jini could not be realized by a Q/A system.

9 Conclusion

This paper demonstrates that publish/subscribe middleware can be used to construct a Query/Advertise system with functionality essentially equivalent to Gnutella. A prototype Q/A system was implemented over the Siena publish/subscribe system to demonstrate the feasibility of this approach.

Further, using a publish/subscribe system solves or mitigates many of the known problems of Gnutella. On the negative side, a number of research problems have been identified whose solution would improve the ability of a publish/subscribe messaging system to support the Query/Advertise paradigm.

Acknowledgements

I wish to thank Alexander Wolf and Richard Hall for their valuable comments about the relationship of Gnutella and publish/subscribe, and special thanks to Antonio Carzaniga for providing Siena.

This material is based upon work sponsored by the Air Force Materiel Command, Rome Laboratory, SPAWAR, and the 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.

References

[1] Eytan Adar and Bernardo A. Huberman. Free riding on gnutella. Technical report, Xerox PARC, 10 Aug. 2000.

[2] A. Carzaniga, D.S. Rosenblum, and A.L. 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] Antonio Carzaniga. *Siena: A Wide-Area Event Notification Service.* University of Colorado Software Engineering Research Laboratory (SERL). http://www.cs.colorado.edu/~carzanig/siena/software/.

[4] Antonio Carzaniga, David R. Rosenblum, and Alexander L. Wolf. Challenges for distributed event services: Scalability vs. expressiveness. In *Engineering Distributed Objects '99*, Los Angeles CA, USA, May 1999.

[5] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Interfaces and algorithms for a wide-area event notification service. Technical Report CU-CS-888-99, Department of Computer Science, University of Colorado, October 1999. revised May 2000.

[6] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Content-based addressing and routing: A general model and its application. Technical Report CU-CS-902-00, Department of Computer Science, University of Colorado, January 2000.

[7] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore 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.

[8] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – http/1.1. Technical Report RFC2616, IETF, 1999.

[9] *Gnutella Home Web Page.* http://gnutella.wego.com/.

[10] *Gnutella Developers Home Web Page.* http://gnutelladev.wego.com/.

[11] *Knowbuddy's Gnutella FAQ.* http://www.lysator.liu.se/ mitja/protest/gnutellafaq.html.

[12] *Jini^{tm} Specification, version 1.1 Beta*, 1999.

[13] Astrid M. Julienne and Brian Holtz. *ToolTalk and open protocols, inter-application communication.* Prentice-Hall, 1994.

[14] Leander Kahney. Intel Says: Think like Napster. *Wired News*, 2000. http://www.wired.com/news/technology/0,1282,38413,00.html.

[15] Declan McCullagh. Get Your Music Mojo Working. *Wired News*, 2000. http://www.wired.com/news/technology/0,1282,37892,00.html.

[16] Paul McDougall. The Power of Peer-To-Peer. *Information Week*, 2000. http://informationweek.com/801/peer.htm.

[17] *Mojo Nation Home Web Page.* http://www.mojonation.net/.

[18] *Napster Home Web Page.* http://www.napster.com/.

[19] *The Working Group on Peer-To-Peer Computing.* http://www.peer-to-peerwg.org.

[20] Steven 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, September 1997.

[22] TIBCO, Inc. *Rendezvous Information Bus*, 1996. http://www.rv.tibco.com/rvwhitepaper.html.

[23] Jim Waldo. Jini^{tm} architectural overview: Technical white paper. Technical report, Sun Microsystems, 1999.

[24] Jason Zien. The technology behind napster. *About*, 2000. http://internet.about.com/library/weekly/2000/aa052800b.htm.

Appendix A Quad Application Programming Interface

```
package quad;
import siena.*;...

public class Quad {
    public Quad(String id, String server) throws QuadException; // Constructor

    ///// Advertising API /////
    public void advertise(Advertisement ad) throws QuadException;
    public int wait_for_query(Advertisement ad);
    public Message get_query(Advertisement ad);
    public void reply(Message query, Message reply, Address myaddr)
            throws QuadException;

    ///// Querying API /////
    public void query(Message q, Address myaddr) throws QuadException;
    public Message get_reply(Address addr);
    public int waitresponses(Address addr, int minresponses);
    public int wait(Address addr, int seconds);
    public void shutdown() {dispatcher.shutdown();}
}

public class Address extends Filter {
    public Address(String id);
    public String address();

public class Advertisement extends Filter {
    public Advertisement(Address myaddr);
    public void definefield(String name);
    public void insert(String name, String op, String value);
    // ... repeat insert for other value types
}

public class Message extends Event {
    public Message();
    public void insert(String name, String value);
    // ... repeat insert for other value types
    public String get(String name);
    public String gettype(String name);
}
```

Appendix B Example Advertising Client

```
package quad.example;
import siena.*;...

public class Advertiser {

public static void main(String[] args) {
  try {
    String id = System.getProperty("quad.id");
    String server = System.getProperty("quad.server");
    if(server == null || id == null) System.exit(1);

    Quad quad = new Quad(id, server);

    Address myaddress = new Address(id);

    Advertisement ad = new Advertisement(myaddress);
    ad.insert("author", "=", "John Steinbeck");
    ad.insert("firstedition", "=", true);
    ad.definefield("title");

    quad.advertise(ad);

    while(true) {
        quad.wait_for_query(ad); // blocks till query arrives
        Message query = quad.get_query(ad);
        //Match the query against the set of available books
        for(int i=0;i<books.length;i++) {
            if(books[i].match(query)) {
                Message reply = new Message(); // generate response
                reply.insert("author",b.author());
                reply.insert("title",b.title());
                reply.insert("firstedition",b.firstedition());
                quad.reply(query,reply,myaddress);
            }
        }
    }
  } catch (Exception ex) {System.exit(1);}
}//main
}//Advertiser
```

Appendix C Example Querying Client

```
package quad.example;

import siena.*;...

public class Queryer {

public static void main(String[] args)
{
  try {
    String id = System.getProperty("quad.id");
    String server = System.getProperty("quad.server");
    if(server == null || id == null) System.exit(1);

    Quad quad = new Quad(id, server);

    Address myaddress = new Address(id);

    Message query = new Message();
    query.insert("author", "John Steinbeck");
    query.insert("title", "Grapes of Wrath");
    query.insert("firstedition", true);

    quad.query(query,myaddress);

    int responses = quad.wait(myaddress,10); // wait 10 seconds

    for(int i=0;i<responses; i++) {
        Message r = quad.get_reply(myaddress);
        // Print the reply ...
    }
    quad.shutdown(); // shutdown cleanly
  } catch (Exception ex) {System.exit(1);}
}//main
}//Queryer
```

Appendix D Output from Example Execution

Appendix D.1 Output from Client Query and Responses

```
java -cp ".../classes/quad.jar:.../classes/siena.jar:." \
     -Dquad.server="senp://localhost:2001" -Dquad.id="queryer" \
     quad.example.Queryer
Queryer: id=queryer server=senp://localhost:2001
Queryer: sending query=message{ author="John Steinbeck" firstedition=true \
                                title="Grapes of Wrath"}
Queryer: begin wait
Queryer: end wait; responses=1
Queryer: received reply=message{ _destination="queryer" _source="advertiser" \
                                 author="John Steinbeck" firstedition=true \
                                 title="Grapes of Wrath"}
```

Appendix D.2 Output from Client Advertise and Respond

```
java -classpath ".../classes/quad.jar:.../classes/siena.jar:." \
     -Dquad.server="senp://localhost:2001" -Dquad.id="advertiser" \
     quad.example.Advertiser
Advertiser: id=advertiser server=senp://localhost:2001
Advertiser: advertisement=advertisement{ _source !="advertiser" \
                                         author ="John Steinbeck" \
                                         firstedition =true \
                                         title any}
Advertiser: waiting for query
Advertiser: query=message{ _source="queryer" \
                           author="John Steinbeck" \
                           firstedition=true \
                           title="Grapes of Wrath"}
Advertiser: matched book={ author=John Steinbeck \
                           title=Grapes of Wrath \
                           firstedition=true }
Advertiser: sending reply=message{ author="John Steinbeck" \
                                   firstedition=true \
                                   title="Grapes of Wrath"}
Advertiser: waiting for query
```