An Efficient Scheme for Query Processing on Peer-to-Peer Networks

Michael T. Prinkey
Aeolus Research, Inc.

DRAFT

Abstract

The peer-to-peer networking paradigm has emerged as an exciting and popular approach for interconnecting individual systems. However, networks that eschew centralized indexing servers have demonstrated a lack of robustness as the node population increases. [1], [2] We present a minor modification to the distributed networking paradigm that allows for an approximate indexing of node and branch content and query routing based on this approximate index information. This modification results in an end to the exponential communication requirements for searches across distributed networks and offers a potential solution to peer-to-peer query scaling problems in general.

Contents

An Introduction to Query Processing on Distributed Networks

The current state of query processing on distributed networks essentially requires that every node in the network process every query. At first look, this feature seems to be an essential part of a fully distributed networking scheme. If we were to posit the existence of a central indexing server, we have surrendered the fully distributed nature of the network. Since each node only knows its own contents, the only way to find something is to ask every node.

Before we continue, let us walk through a simple query process in a tree network (See Figure 1). Nodes in the network connect up to a host node and down to hosted nodes. Query relaying on tree networks is trivial. All messages are relayed up to the host node and down to all hosted nodes.

FIGURE 1: A Prototypical Tree Network

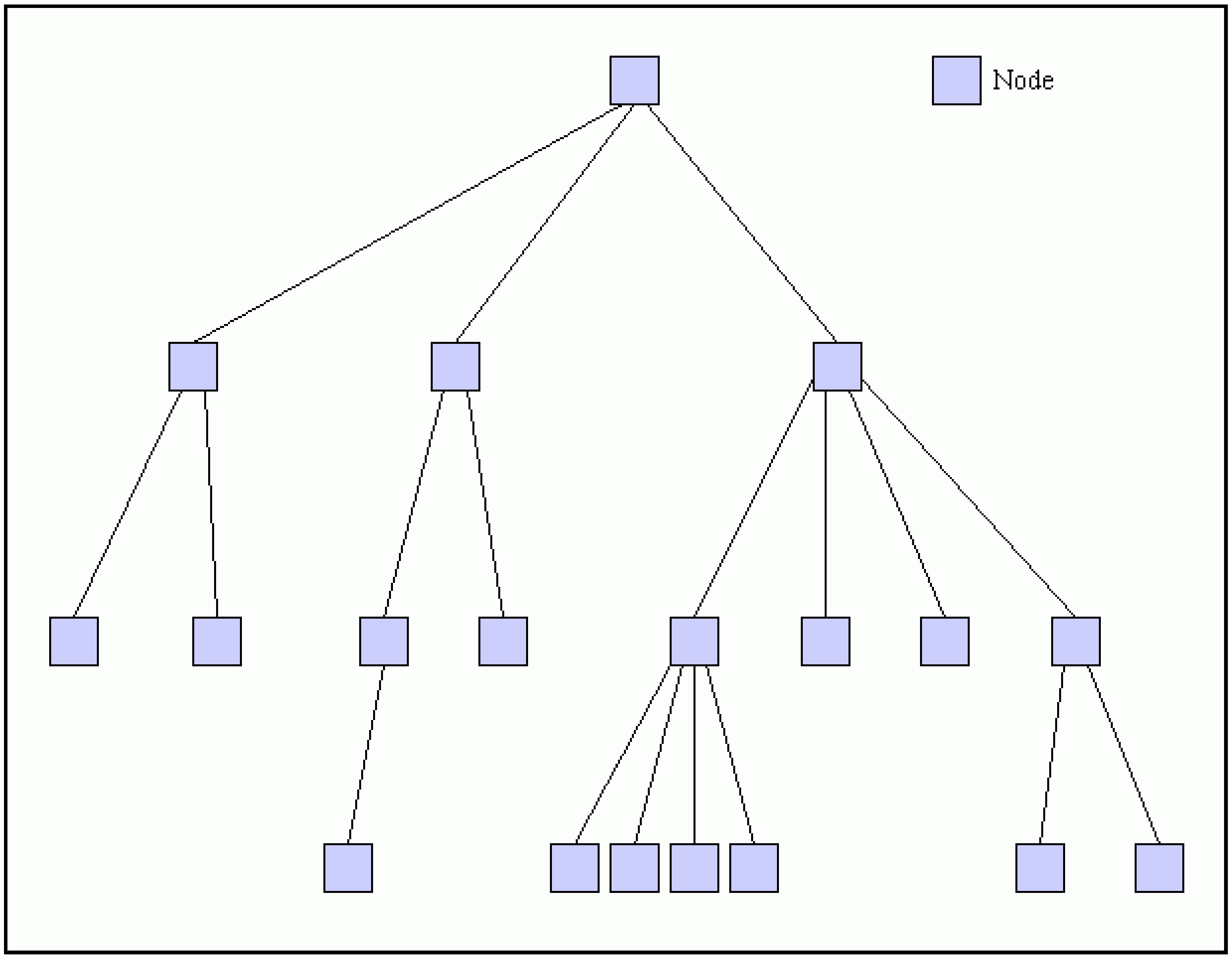
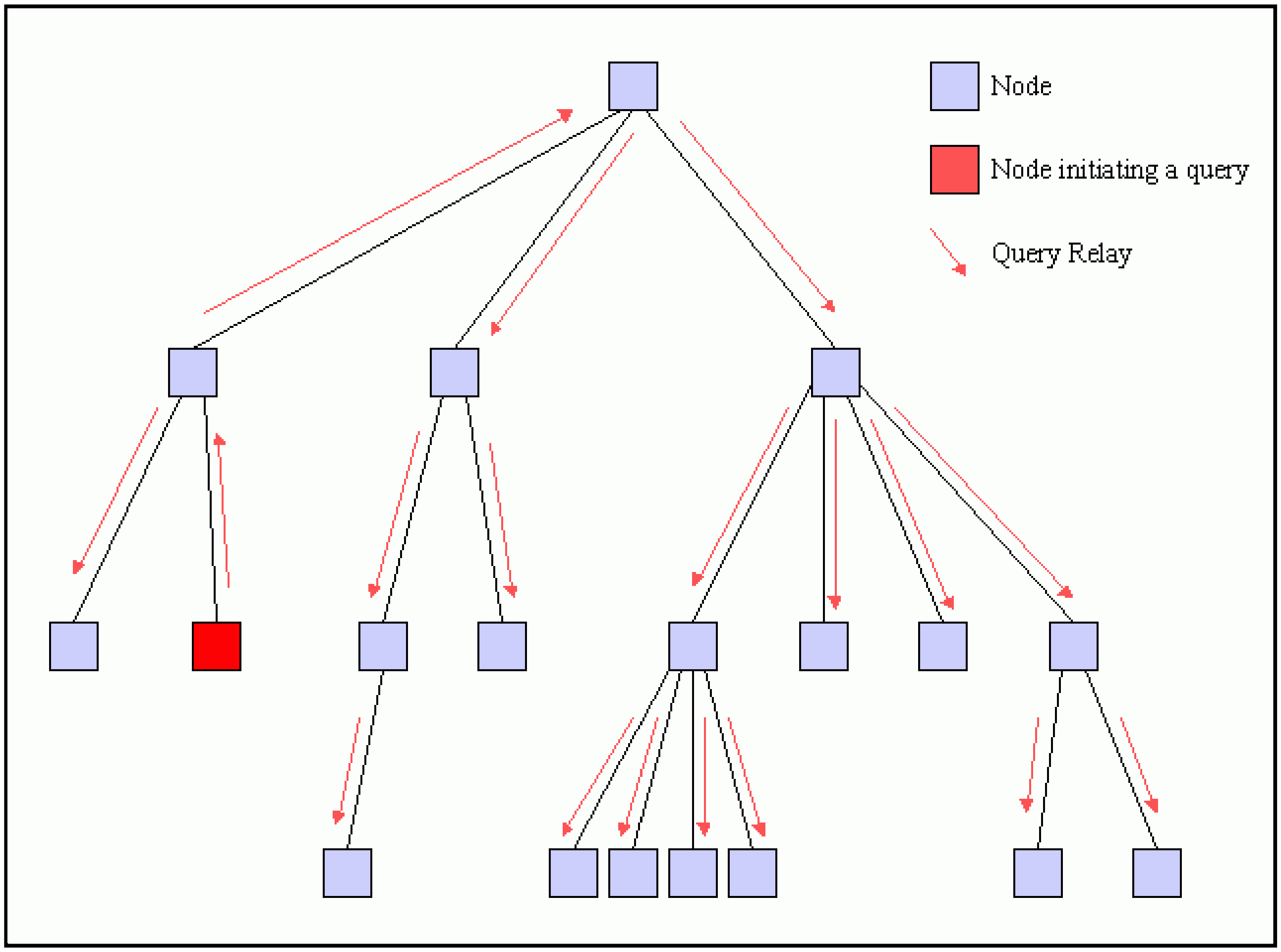


Figure 2 illustrates the normal query process. The query originates at a node and is forwarded up to its host. The host uplinks the query and forwards it down to other nodes that it is hosting. The process continues recursively until the entire network receives a copy of the query.

FIGURE 2: Diagram of a Query Relay



It is easy to see that the total communication load inflicted on the network grows as a function of the network population. It is also easy to see that there exists a fundamental limit to the number of queries that may be processed by a fully distributed network in terms of bandwidth requirements of the slowest nodes. If a query requires Q bytes and the slowest links have bandwidth B bytes/sec, that node can only process B/Q queries per second. If a node in a tree topology cannot process the query, it effectively truncates that branch from the tree and results in network fragmentation.

This is a fundamental limitation in peer-to-peer networking. It cannot be superseded by compression algorithms or more clever protocols. If every machine is responsible to handle every query, the network paradigm is doomed to suffer this limitation irrespective of the available bandwidth. There is an analogous problem in other areas of network technology. We need to learn from their solutions.

The Routing Paradigm

When local area networking, and specifically, 10Base2 Ethernet (coax) became popular, the 10 Mbps network bandwidth provided a very useful communications medium for systems of the time. But, as Ethernet LANs became larger and larger, the shared bandwidth on the network became limiting. Only one system could "talk" to the network at once. Every system on the LAN "heard" it speak, but usually only one system was "listening." The solution to the performance limitations imposed by the shared bandwidth was to segment the network using routers.

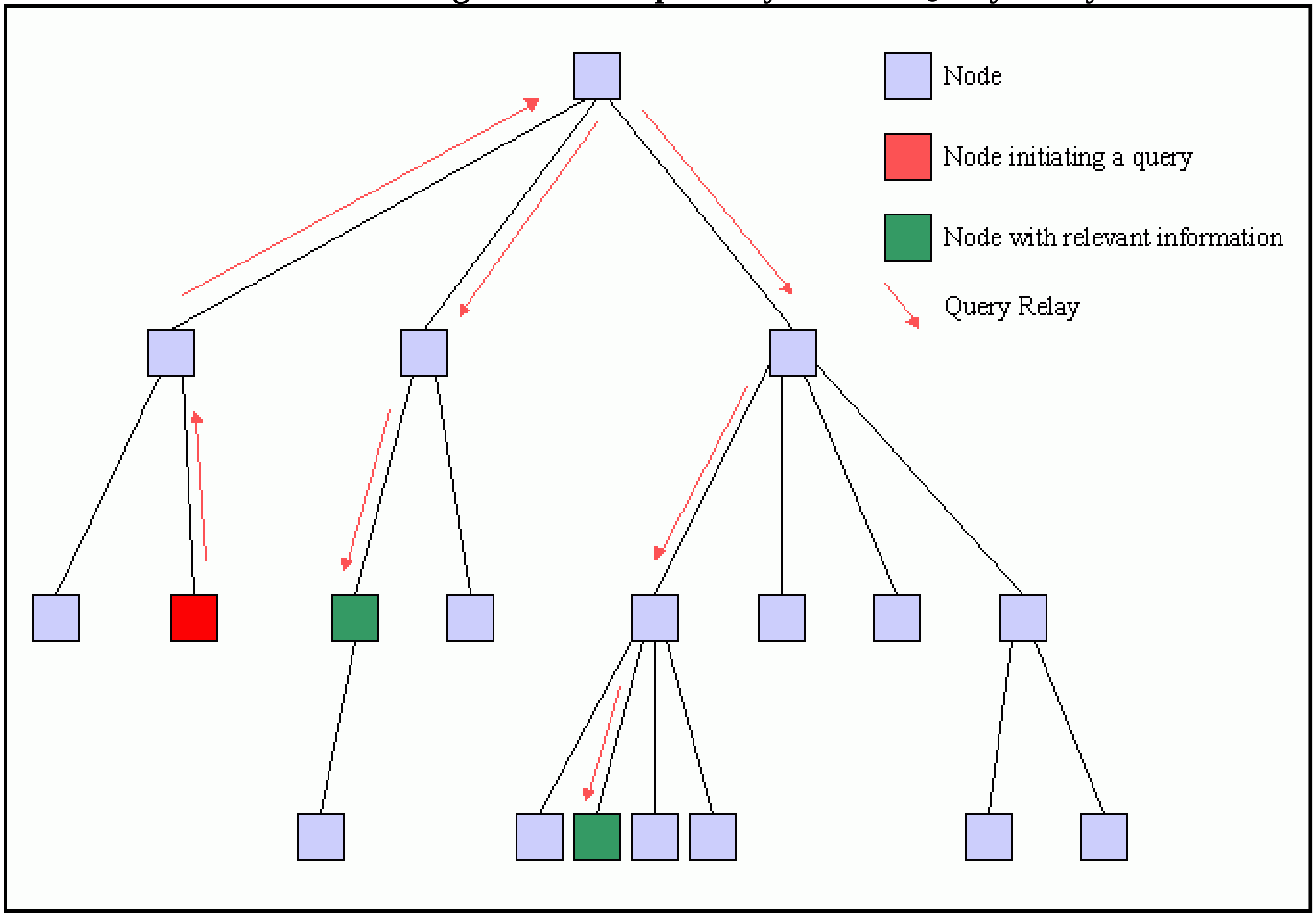
Routers are not terribly complicated systems, in theory. Their job is to ascertain which directions (if any) to send incoming information. By using a routing table, the system sends packets onto the

appropriate network segment and does not clutter the other network segments with packets that it knows will not find a recipient. High-performance router technology is one of the main reasons that LANs and the Internet in general are as efficient as they are currently. If every system on the Internet had to "listen" every time someone sent an email or downloaded a web page, the system would be completely useless.

That last statement is the key: *A network where every system is listening is doomed to become useless.* This is the lesson that we should learn for peer-to-peer networking. And it begs a question: How do we prevent all of the systems from listening to every query? Essentially, how do we route queries?

Consider the optimum route that a query should take. A given query should only touch those nodes and branches that lead directly to the requested information. An optimal query route is illustrated in Figure 3. We see that the query travels up through the tree and is only directed down to branches and nodes that contain useful information. The total number of query relays, and hence the total communication burden on the network, now scales with the number of nodes containing relevant data, not with with the overall population of the network.

FIGURE 3: Diagram of an Optimally Routed Query Relay



There is significantly less communication required to convey the query if we do not forward the search to every node. Query routing can alleviate the traffic problems experienced by low-bandwidth nodes and prevent network fragmentation. A quiet network is a happy network.

Of course, for an optimal query path to evolve, every node would need to know the complete contents of

all of its connected nodes. Collecting, updating, and searching all of the information in all connected nodes and branches would be a significant burden on all of the nodes in the network. The dynamic nature of the network and the data it contains makes this approach unrealistic. Furthermore, if each node possessed complete information for all nodes, then any node could complete the query itself. In a sense, Gnutella's Reflector application [3] does this for a selected subnet. It does process queries on behalf of its nodes and so insulates those nodes from the bulk of query traffic. In essence, the Reflector builds a network segment which allows nodes on low-speed connections to participate in the network. Though this is an admirable advance, it fails to address the scaling issue for the network as a whole. The system running the Reflector is still subject to bandwidth limitations as it must process every query on the network. The bandwidth B in our previous equation goes a bit higher and hence the total number of queries on the network can be a bit higher before the network begins to come apart. But, therein still lies an inherent scaling limitation for the network. As the network grows and system-wide query rates increase, eventually bandwidth on the Reflector nodes will be insufficient and the network will fragment.

But our stated goal is merely to route the query, not to resolve it exactly. For this, it is not necessary for complete query processing to be handled by proxy. It is really only necessary for irrelevant queries to not be forwarded to nodes or branches. But, how do we know that a query is irrevelant without checking the node or branch contents?

We cannot know precisely if a query is relevant without complete information, but it may be possible to approximately process the query using a reduced information set and route the query based on that approximation. An approximate query test may lead to false positives which will cause queries to be forwarded to nodes or branches that do not contain relevant information. Essentially, the approximate query tests will lead to sub-optimal routing, but as long as the false positive error rate is small, we will have queries routed along nearly optimal paths and most of the nodes that finally receive queries will in fact contain relevant information. Furthermore, we can design the approximate query scheme to guarantee that there will be no false negative responses, so no nodes will be overlooked and thereby ensure that the fidelity of the query remains intact.

Indexing Schemes and Approximate Query Processing

A great deal of literature exists on the acceleration of search processing. Don Knuth's *The Art of Computer Programming, Volume 3* [4] devotes several hundred pages to the topic. Unfortunately, the algorithms were constructed for centralized databases. In the case of peer-to-peer networks, the database that must be queried is distributed and highly dynamic. Nevertheless, there are many features of these query algorithms which we can borrow to build a search scheme for a distributed database.

One of the key approaches to high-speed querying of a database is the construction an index. Essentially, the index is a highly compressed representation of the data which can be searched much more efficiently than the entire database. One very useful indexing scheme relies on a hashing function to reduce text keywords into a small set of integers. The index will then consist of a list of these integers. Searches can be conducted by simply searching the index rather than pouring through the entire database. We have no intention of revisiting all of the details of hash function construction or indexing schemes, but we will review some of the basic concepts to provide a framework for the construction of a distributed query algorithm.

Let us consider a very brief example. Assume that our database contains the following sentence fragment:

```
When in the Course of human events, it becomes necessary for one people to dissolve the
political bands...
```

The indexing scheme would typically discard common words like `when, in, becomes, the, it, for, to`. This leaves several keywords: `course, human, events, necessary, one, people, dissolve, political, bands`. Furthermore, we will treat the keywords as case insensitive. If we assume that we have a hashing function which casts our keywords into an integer between 0 and 99999, we can assemble a portion of the database index corresponding to this fragment:

course	= 02123
human	= 10932
events	= 34394
necessary	= 50234
one	= 68023
people	= 43020
dissolve	= 10593
political	= 42330
bands	= 92399

Now, rather than searching the entire database for a given keyword, we can simply search the index for its corresponding hash value. For the example above, we could find `political` by searching the index for 42330. And likewise, we could fail to find `nation` by searching the index for its hash value, 14092.

Hashing-type indexing schemes are commonly applied to individual portions of a database. We might build an index on a per-page or per-paragraph basis for a long document. We could then quickly search the indices for each portion of the document and identify which portions contain relevant data, or more to the point, ignore the many portions that contain nothing of interest. It is this same approach that we would like to borrow in order to facilitate the search of distributed databases. We could apply a similar hashing function to the metadata for shared information contained on each node in a peer-to-peer network. This index will encapsulate the data contained on each node.

Typically, index construction with hashing functions requires that we examine the results to identify collisions (instances where two or more keywords hash to the same value). If no collisions occur, we have a perfect hash. If collisions do occur, some collision resolution mechanism must be implemented. Collisions can be very common if we choose too small of an index space. For our earlier example, had we chosen a hash function which mapped the keywords into integers ranging from 0 to 7 (3-bits), the index could look like this:

course	= 4
human	= 7
events	= 0
necessary	= 1
one	= 5
people	= 7
dissolve	= 6
political	= 3
bands	= 5

Since we have nine keywords and only eight index states, at least one collision is unavoidable. In this case, we have two collisions--for hash values 5 and 7. Note that hash value 2 does not appear at all. So, by using this index, we can find false positive results to our query test. As far as this index is concerned:

```
human == people
 one == bands
```

If we choose a good hash function, the distribution of results across the hash space is highly random based on the input. So, in general, we will see fewer collisions in larger hash spaces. More specifically, doubling the size of the hash space will (on average) half the number of collisions.

Unfortunately, we can neither test for collisions or provide for collision resolution. This is because the

distributed and dynamic nature of the data precludes us from performing these tests in a timely manner. Because we cannot resolve the query exactly, we must treat this as an approximate method. And, by increasing the size of the hash space, we have a mechanism to increase the accuracy of the approximation.

Luckily, the routing of queries does not require absolute accuracy. We only need to identify whether or not a node (or branch) could potentially satisfy a query, and not that it satisfies it positively. We will use this approximate indexing approach to cull the node population and route queries. Let us emphasize that the approximate nature of the routing scheme will not affect the fidelity of the final query. The nodes which finally receive the routed queries will ultimately decide whether or not it contains something relevant. The approximate query processing simply avoids sending the query request to systems that cannot satisfy the query.

Content-based Query Routing

So, the hashing of metadata keywords provides us with a signature [5] which we can use to determine if a node possibly contains information relevant to our query. This is valuable. If each node's host had this information, it could determine whether or not to send each incoming query to that node. This is potentially useful, but it only takes care of the "last hop" in the network. This allows us to route queries at the bottom of the tree. In a sense, this would to a large degree duplicate the functionality of Reflector. Queries would only be sent to hosted nodes if there is a high probability that the node contains relevant data.

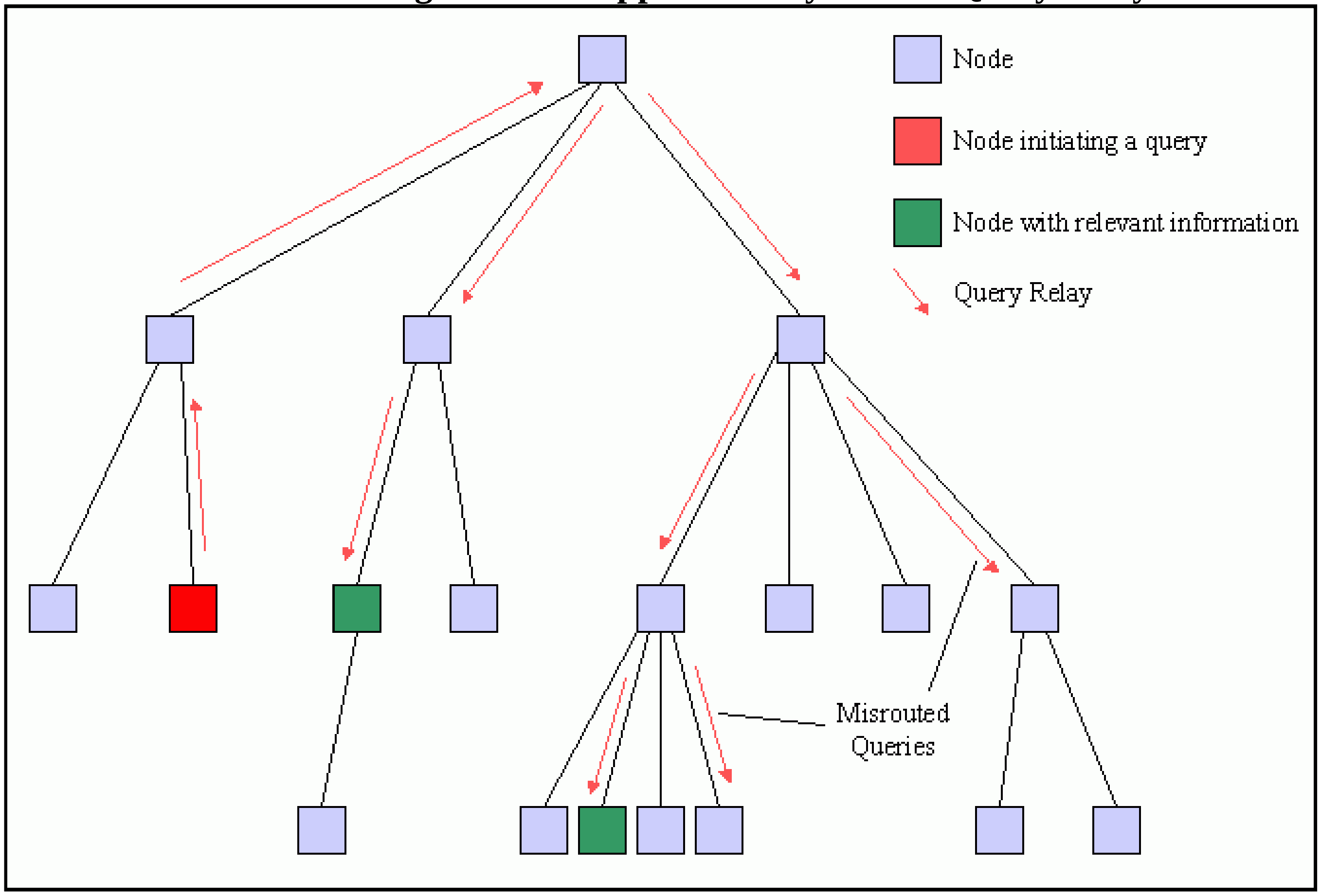
While this is useful, it does not represent a significant leap forward. Removal of the last hop is not the best that we can do. Again, we will borrow both a concept and terminology from routing approaches. We will use a bitmask to represent our hashed index space. If we choose to have 2^{18} entries in our hash space, we can represent our node signature as a bitmask comprised of 2^{18} bits. The bit will be set if the corresponding hash index is present on our node and off is it is not. In this instance, our hash space would contain approximately 256,000 entries and would require only 32 kilobytes in its raw form. Special encoding approaches could reduce this size significantly. We will address this in a later section.

This signature must be passed up to the host node. The host node remembers each of the signature bitmasks for its hosted nodes. Then it takes the bitmasks from its own index and Logically ORs them with all of the bitmasks from its hosted nodes. This aggregate bitmask provides a signature of all of the information in the *entire branch*. This aggregate bitmask is then passed up to its host. The approach can be applied recursively up the tree.

With this information in hand, queries can now be routed at every level of the network. If a hosted node or branch does not contain all of keywords in an incoming query, it is not routed there. This approach is completely scalable. Bitmask aggregation will continue up the tree and will essentially enumerate all of the information contained in each branch and sub-branch. Most importantly, note that the bitmasks do not grow. Each will be no larger than 32 KBs. This size limitation is key to preventing the higher level nodes from being overwhelmed with index information.

So, now let us follow a query through the tree network shown in Figure 4. One node originates the query. By assumption, the query is always sent up to the host node. The query is then tested against the bitmasks for each of the connected nodes/branches. If they pass, then the query is forwarded. Otherwise, nothing is sent. The host node receives the query and also forwards it up it its host. It then tests the query against bitmasks of its connected nodes/branches and forwards the query as needed. On average, only a small fraction of the nodes/branches will pass this culling procedure, so only a handful of queries will be forwarded. Only those nodes lying on branches which potentially can satisfy the query will receive it. The rest are left alone.

FIGURE 4: Diagram of an Approximately Routed Query Relay



The thought experiment outlined above describes an approach that we have termed Content-based Query Routing (CQR). By using the index-based bitmasks approximately representing node and branch content, nodes can make intelligent decisions about the forwarding of queries rather than simply broadcasting the query to every node. If implemented properly and integrated into peer-to-peer networking protocols, this approach could lead to the elimination of the fundamental scaling problems inherent in these networks.

There are three key parts to CQR. The first is the approximation of the content of nodes. The second is the aggregation of those per-node content approximants into branch approximants and distribution of this information to host nodes. And the final part is the routing scheme itself. In the next section we will address the types of approximation errors that can occur and attempt to estimate their effects on the network. For now, we will not address the routing issues. Routing within a tree network is trivial and does not require discussion; however, the CQR principles may be applied to a more generally connected network.

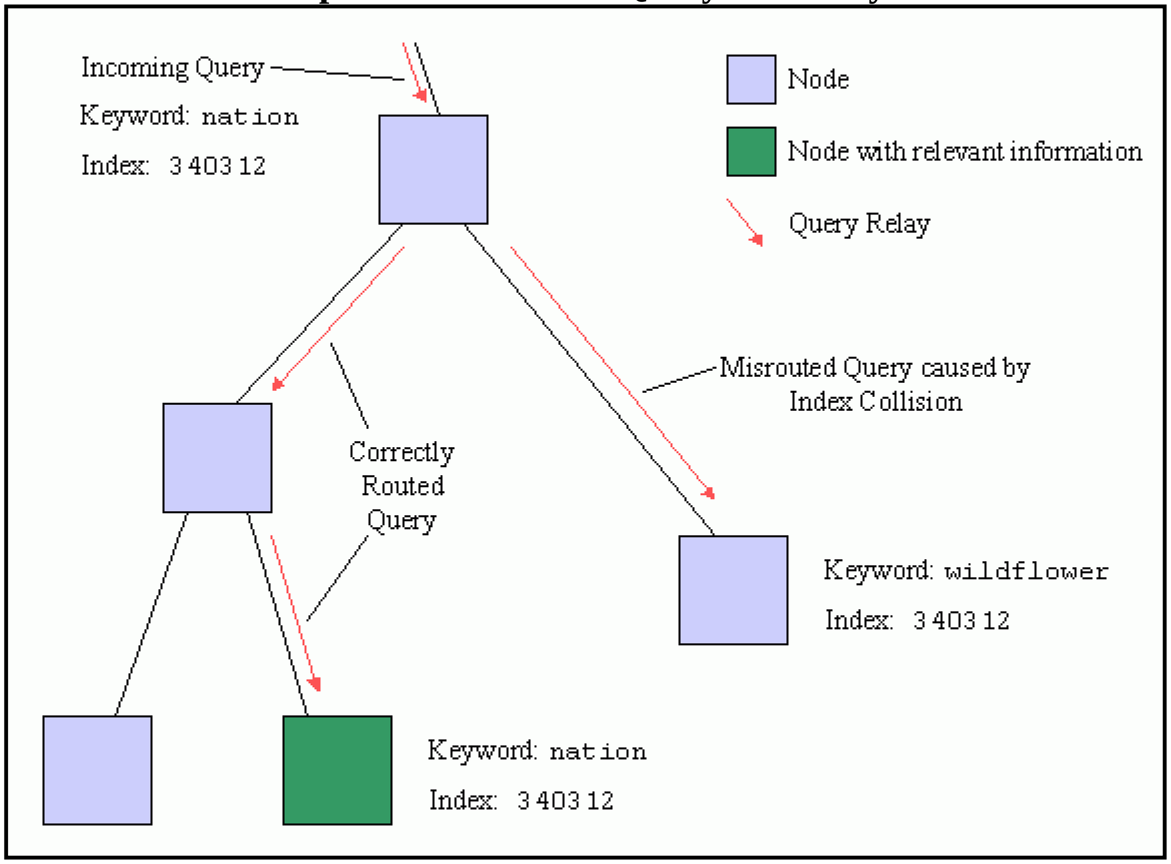
Assessment of Query Approximations

As we discussed earlier, it is impossible for our index-based query scheme to return a false negative response. If one or more of the queried keywords do not exist in the index of a given node or branch, then that node or branch cannot satisfy the query. However, two types of false positive responses can occur. These false positives from the approximate search will lead to misrouted queries. False positives can occur for two reasons: collisions in the hash space and keyword aggregation.

The concept of hash collisions has already been addressed. Essentially, we need to choose a large enough space (and corresponding bitmask) to result in a small collision rate. There is a trade-off to be made here. We could choose to cast the keyword space into an unsigned long integer with range from 0 to 2^{32}-1 (about 4 billion). Such a large index space would give us a very low probability of keyword collisions (e.g., there are only about 300,000 words in the English language). However, our bitmask would then be over 500 MBs in size! This is far too large to be useful. For pragmatic reasons, our index space will need to be limited to 2^{18} to 2^{20} entries which will lead to 32 KB to 128 KB bitmasks, respectively. We will simply accept some keyword collisions and misrouting as part of the algorithm. Note that misroutings of this type will trickle down to the node that contains the collision keyword. In that sense, it has the more deleterious effect on the network. But for a reasonably sized index space, these errors should be infrequent.

Figure 5 shows the diagram of collision-induced query misroute. The incoming search contains the keyword `nation` which has a hash index of 340312. The host node properly routes the query to the left branch and the query is satisfied. However, a misroute occurs when it also sends the query to the right. That node does contain index 340312, but this hash arises from the keyword `wildflower`. Ultimately, this node is not able to satisfy the query even though the hash indices match.

FIGURE 5: Example of a Misrouted Query caused by Index Collision

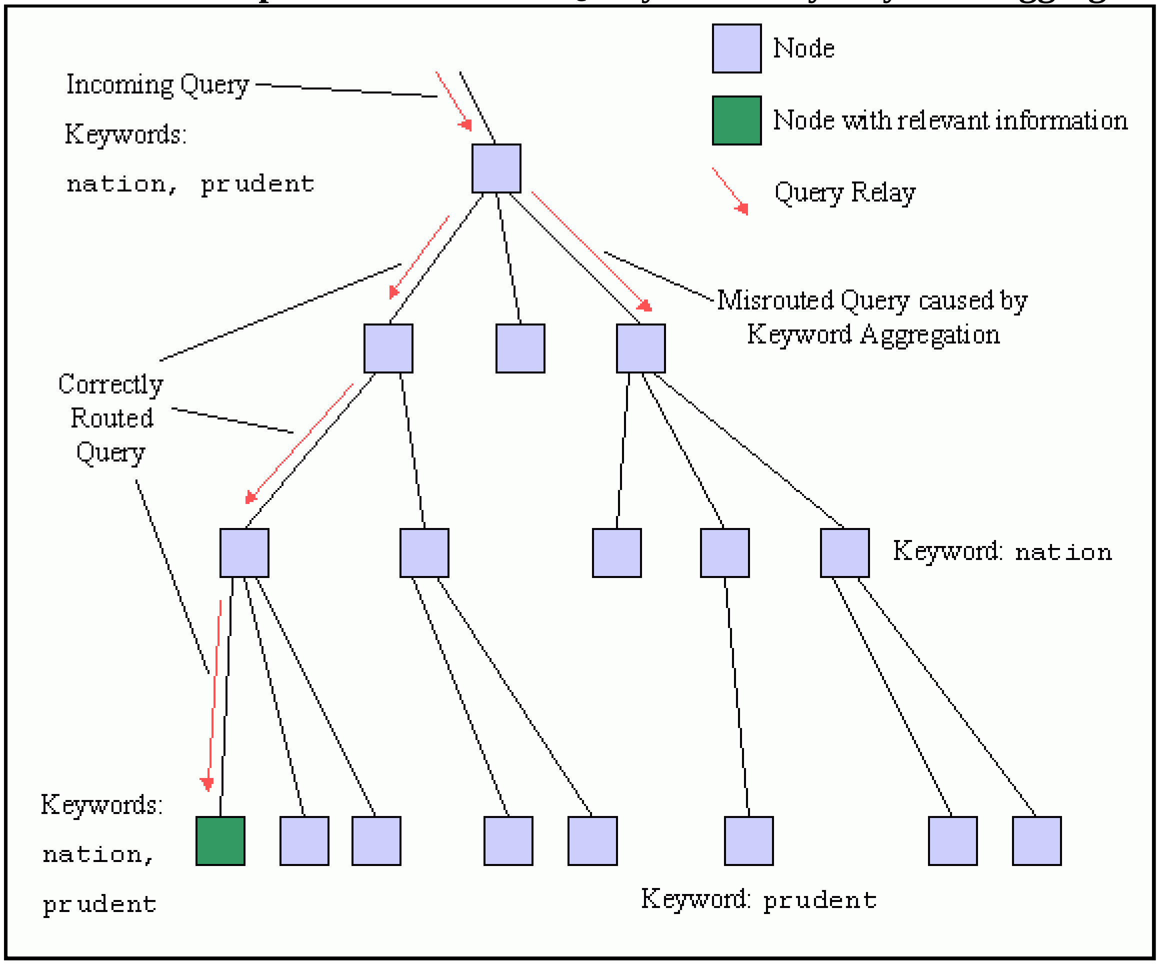


The second type of routing error arises from the aggregation of separate node bitmasks. For searches which contain more than one keyword, it is possible that several nodes in the branch contain individual keywords, but that no one node contains all of them. This type of misrouting will be more common, especially for branches containing many nodes. The aggregate bitmasks for the systems near the top of the network will be very "full." Fortunately, misrouted queries of this type will only descend partially into the branch. As soon as there is a "split" where branches no longer contain all of the keywords, the

misroute will end.

Figure 6 illustrates a misrouted query arising from keyword aggregation. The incoming query contains two keywords: `nation` and `prudent`. Again, the query is correctly routed to the left branch and trickles down to a node that contains both keywords. The query is incorrectly routed to the right branch because the aggregate bitmask contains the indices for both keywords. Each keyword is present on separate nodes in the branch. The query only progresses down one level into the branch before the keyword split occurs. The keyword `nation` is contained in the sub-branch to the right while the keyword `prudent` is contained in the sub-branch to the left. The query is not routed further.

FIGURE 6: Example of a Misrouted Query caused by Keyword Aggregation



Unfortunately, there is no easy way to maintain a fixed bitmask size and avoid aggregation-type collisions. Any other technique will accumulate information as the bitmasks are passed up the tree. For example, rather than using Logical OR to combine all of the bitmasks, the host could instead uplink all of the bitmasks. This is tantamount to sending full list information for each node, and will result in a significant bandwidth burden for nodes near the top of the tree.

Rather than uplinking all of the information, we can try to leverage the random nature of the hash function to help preserve the cross-correlated information from each node. For a large index space, a given node will yield a very sparse signature bitmask. It is likely that only a few thousand keywords will be present on any given node. The signature creation algorithm can be modified to include a bit-rotation following the normal hash function. The final hash bitmask would be rotated by a fixed number of bits based on an unique key for that node. This key can be generated randomly on each

node. It would be an unnecessary complication to have the host assign this key as the key space is sufficiently large that randomly generated keys will almost never intersect. Note that the key would have to be the same size as the number of bits in the bit mask. This key would need to be uploaded to the host along with the normal bitmask. So now, two nodes can contain the same key word, but they will encode to different bits in the aggregate bitmask.

The host will collect the bitmasks from each connected node as before. It will generate its own bitmask and then rotate it using its own key. These separate rotated bitmasks can then be combined using Logical OR. This key along with the compiled list of all keys must be uplinked to the next higher level. This does cause the amount of uplinked information to increase somewhat, as all of the unique keys must be propagated up the tree, but the key size will be only a 2^{18} to 2^{20}-bit number and hence will only require 2.25 to 2.5 bytes to encode each of them. Even if 4,000 machines are connected to a given branch, this will only add 10 KBs to the data signature at the top of the tree.

Now, the host must remember the bitmask and the key for each node in its branch. When a query comes, it must first apply the normal hash function to identify the unrotated index and then rotate the hashed result for each node based on its unique key. The same bitmask comparison technique is used. This will increase the amount of effort required to ascertain whether or not a query succeeds for a given branch, because there is a separate rotation for each node in the branch.

This approach, while preserving the cross-correlation information for individual nodes, does so at the expense of the hashing scheme. For a given node, the hash bitmask will be very sparse. It is likely that many nodes in a given branch will contain several of the same keywords. But, since each index table is rotated, these common keywords will not "overlap" but will instead occupy different bit locations. The bitmask will fill much more quickly, as is required to preserve cross-correlation information. The result is that hashing/rotation collisions will be much more common than the simple hashing collisions were before. Misroutings can still occur, but in more subtle ways. In essence, the index collision errors and keyword aggregations have merged. However, using this approach, it is possible to maintain a nearly constant signature size and remove keyword aggregation collisions. We can reduce the probability of all types of misrouting by increasing the size of the hash space. It is an open question whether this is worthwhile or not.

If this approach is taken, great care must be exercised in the removal of common words prior to hashing. These common words like `and, the, in...` will appear in virtually every node. But with the bit rotation (or any per-node supplemental hashing scheme) applied, each of these common words will occupy a different bit location on each node. Hence, they will populate the aggregate bitmasks very quickly. If there are 50 common words, a branch with 100 nodes will "waste" about 5000 bit locations, or roughly 2% of a 2^{18} bitmask. Furthermore, the bitmask must be chosen large enough to allow for the inclusion of the cross-correlation information. In this modified approach, keyword collision probabilities will scale roughly with the number of nodes in the branch, hence the need for a larger bitmask. Finally, the uplinked bitmasks will be necessarily less sparse by the inclusion of the per-node bit rotation. This will decrease the efficiency of encoding schemes that leverage sparsity. The maximum size of the bitmask will remain fixed, but the size of the uplinked data may be larger because of the loss of coding/compression efficiency.

It is not at all clear that the extra complexity of this second approach is warranted. It is likely that index collision errors will be infrequent and that keyword aggregation errors will not be a significant burden as they will usually fail to propagate far into the network. However, it is impossible to test these various approaches without using real data. We recommend only applying this more complicated approach if the simple implementation proves unacceptable.

Implementation Issues and Protocol Modifications

What we have outlined thus far is an overarching approach to distributed database query processing. The introduction of hash-based bitmasks allows a signature of each node's data to be propagated up the network tree where it can be used determine whether or not that node should receive a given query. Furthermore, these bitmasks can be aggregated and further uplinked to allow hosts above to ascertain whether or not to route the query down the entire branch. Now, we need to outline a general implementation plan which will allow peer-to-peer networks to implement this approach.

All peer-to-peer clients necessarily know how to index their shared information. Furthermore, they already contain some protocol to attach to host computers. The implementation of CQR requires the selection of a universal hashing scheme, the handshake protocol to facilitate the uplink of the resulting bitmask, a sane keep-alive schedule for the updating/deletion of the bitmask, and an appropriate encoding scheme to reduce uplink data requirements.

A great deal of time and effort can be devoted to the selection of the hashing function. And, in all likelihood, that optimal hashing function will probably be only incrementally better than one chosen from the normal stock. Knuth's book offers some recommendations. Others are commonly available. Since the keyword space is largely unknown, making the choice is mostly guesswork anyway. The only requirements are:

- **It should handle arbitrary length strings.**
- **It should fully utilize an arbitrarily sized index space.**
- **It should be reasonably fast and completely platform independent.**

To make the CQR implementation fully backward compatible with existing protocols, the uplinked bitmask can be optional. Unfortunately, this means that the default bitmask for every non-CQR node will be full. This also means that every uplinked bitmask will also be full. This will be a potential barrier to acceptance of the new protocol as a single "old" client connecting to a host will fill the bitmask for the entire branch and essentially ruin CQR. It may be necessary to "zero" out the bitmask for old clients to prevent this from happening. This is as much a political issue as a technical one. We mention it only to call attention to the issue, and not to offer a solution.

The handshake protocol can allow the node to quickly connect to the network and then accept the bitmask after connection. Until the bitmask is uploaded, the host will work with a null bitmask. After the bitmask has been compiled and uploaded to the host, the host will recompile the aggregate bitmask and uplink it. To keep the update process sane and prevent excessive uplink traffic, there should be a several minute keep-alive protocol. For now, assume a 10-minute keep-alive timer. If a node does not check in every 12 minutes, it is disconnected. We recommend the following approach:

- **When a new hosted node connects to our node, it consults it keep-alive timer. If it has been more than 5 minutes since the last uplink, it will compile a new aggregate bitmask and upload it to its host immediately. The host will respond, setting the next keep-alive limit, and our node will reset the timer.**
- **If it has been less than 5 minutes since the last uplink, the upload will wait until the next scheduled update.**
- **If nothing has changed, our node needs only to send a "No Change" message to its host.**
- **The host will attempt to synchronize updates from its nodes to occur 1 or 2 minutes prior to its scheduled update. This will prevent delays in the proprogation of new connected data up through the network. Synchronization information can come in the form of a modification to the keep-alive timer. When the host responds to a hosted node, it can adjust the next scheduled keep-alive limit appropriately.**

This approach will prevent excessive upload traffic on busy nodes and will attempt to minimize the time required for a node to come "online" and become visible to the rest of the network.

Finally, we want to mention that the sparsity of the bitmasks compels us to use run-length encoding or direct index enumeration to improve data efficiency. These approaches can lead to a much smaller representation for the bitmask and reduce the upload data size from 10s or 100s of KBs to a few KBs. Of course, if these schemes fail to reduce the size of the representation, the raw bitmask can be sent instead. It would be best to allow for both approaches so that sparse bitmasks can be encoded and full bitmasks can be send in raw form.

Conclusions and Future Work

We have attempted to outline a significant improvement to the peer-to-peer networking paradigm which we hope will facilitate search processing. Moreover, we hope that this modification will be implemented and will improve the robustness and general utility of existing and developing networks.

We have tried to examine the consequences that Content-based Query Routing could have on a peer-to-peer network. We are convinced that the effect will be very positive and will result in a significant reduction in overall query traffic and network fragmentation. Perhaps even more interesting is the prospect of moving away from a tree networking topology. With a routing scheme in hand, it may be possible to allow systems to connect and reconnect in a more general network topology. Then, some of the real genius of optimized routing schemes could be brought to bear.

References

1. **Bandwidth Barriers to Gnutella Network Scalability, http://dss.clip2.com/dss_barrier.html .**
2. **Why Gnutella Can't Scale. No, Really, http://www.darkridge.com/~jpr5/doc/gnutella.html .**
3. **Reflector Overview, http://dss.clip2.com/reflector.html .**
4. **Knuth, Donald, *The Art of Computer Programming, Volume 3: Sorting and Searching*, Second Edition, Addison-Wesley, pp. 513-558, 1998.**
5. **Faloutsos C., "Signature files", In: Frakes W.B., Baeza-Yates R. (eds.) *Information Retrieval Data Structures and Algorithms*, Prentice Hall, New Jersey, pp. 44-65, 1992.**

(C)opyright Michael T. Prinkey, Aeolus Research, Inc. 2001.