Case Study: Gnutella

Searching Algorithms in Unstructured Peer-to-Peer Networks

Ruixuan Li
College of Computer Science, HUST
rxli@public.wh.hb.cn
http://idc.hust.edu.cn/~rxli/

In Courtesy of Xiaodong Zhang, Ohio State Univ
What is Gnutella?

• Gnutella is a fully decentralized peer-to-peer protocol for locating resources

• **Standard**, not a program. There are many implementations of Gnutella (BearShare, LimeWire, Morpheus)

• Each member of the network may act as a client, looking for data, or a server, sharing that data

Server          Client

Servent
Overlay Networks and Gnutella

- A network on top of a network
- Used only for the search aspect of Gnutella
- Actual transfer of files is not performed with the overlay network
Connecting to the Network

• When you start the application, need to first connect to a node - bootstrapping

• No set way to do this. Protocol is silent on the issue
  – Node Cache – contact a well known node that maintains a list of nodes
  – Node Web Page – lists locations of nodes to bootstrap and get started
A bit about TTL and Hops

- TTL and Hops are fields in a Gnutella message that are used to control the diameter or horizon of queries.
- Except when used as a flag in special cases, as you move forward, you decrement TTL and increase Hops.
More about TTL and Hops

- In general, you greatly increase the nodes you can connect to as you broaden your horizon.

\[ n_{hosts} \approx \sum_{i=1}^{n_{hops}} (d-1)^i \]

- Where \( n_{hosts} \) is the number of hosts reachable, \( n_{hops} \) is the number of hops, and \( d \) is the average number of hosts reachable from each host (degree of connectivity) (assumes acyclic graph)
Ping and Pong

• After knowing where to go to join a network, a servent uses **Ping** and **Pong** to locate other nodes to connect to

• Originally, **Ping** message sent to the node connected to, which forwarded to all neighbors

• All those servents willing to connect to the sender respond with **Pong**

• Pongs convey IP, Port, Files shared, and distance (hops)
Pong Caching

• Problem with this approach was traffic – broadcasting causes too much traffic
• Newer standard uses “Pong Caching”. Servents keep information about nearby nodes, and send this when Pinged
• Protocol suggests one method for how to maintain the cache, but doesn’t require that it be used
Pong Caching, Continued

- Use modified TTL and Hops
  - TTL=1, Hops=0 is a probe – servent only responds about itself
  - TTL=2, Hops=0 is a crawler – servent responds with pongs from immediate neighbors only
Pong Caching, Continued

– TTL=7, Hops=0 is a probe – servent replies about any nearby nodes
  • Protocol suggests 10 pongs should be returned – those 10 that have most recently been received
  • Must be good pongs (recent responses, well behaved servents)
  • Servents should send every 3 seconds, but responses come from other’s cache, so bandwidth is claimed to be low
Queries in Gnutella

- Queries work very much like the original Ping/Pong scheme
- Send a **Query** to neighbors, with TTL=7 and Hops=0
- This defines the default “horizon” for reaching servents. As TTL increases, so does the horizon, and the number of hosts that can be reached, improving odds of a hit
Queries in Gnutella, Continued

• Send **Query** to neighbors, who send to their neighbors
• Decrement TTL, increase Hops when you receive
• Continue forwarding while TTL != 0
• If you have the requested file, return a **Query Hit** message
• Generates lots of messages. Called “flood” search
Transferring Data

- When a file is located using the Gnutella protocol, the Query Hit message returns a host/port to contact to obtain the file
  - The connection to get data is made using HTTP
  - Usually (but not always) same host/port as used for Gnutella traffic

- Problems occur when servent is behind firewall
Firewalls and Push

• Gnutella protocol can usually pass through firewalls if servants send/receive on same ports – Ping/Pong mechanism will keep pinholes open

• Problem comes with direct connections from remote host
Flow Control

• In order to control network traffic, Gnutella suggests a very simple flow control method
  – Servents create a FIFO output queue
  – When half full, enter flow control mode, stay as long as more than one quarter full
    • Drop all input
    • Sort queue to send responses before requests
    • Long traveling responses gain priority, Long traveling requests lose priority
Bye

- Message to tell other clients you have terminated
- Not required, but recommended
- Should include a reason code. Loosely defined in the protocol
Format and Style of Gnutella Messages

- Connections are made via TCP
- Classic HTTP style text triple handshake – CONNECT, replied to with 200, replied to with 200
- After this, Gnutella messages are exchanged
  - Gnutella messages are binary
- Each requests uses a semi-unique ID number
Problems with Flood Query

• Traditional Gnutella flood query has a number of problems
  – Very large number of packets generated to fulfill queries
    • Study [2] indicates that most searches on Gnutella can be satisfied with a search that visits fewer nodes
    – Essentially, just a Breadth First Search (BFS)
  – Some proposals [3,4] attempt to address this with alternate schemes for searching
Alternatives to Flood Query

- Iterative Deepening
- Directed BFS
- Local Indices
- Random Walkers
Iterative Deepening

• Essentially, idea is multiple, sequential breadth-first searches

• First, search to a shallow depth

• If no results are returned, repeat search, but go to deeper depth
Iterative Deepening

• To implement more efficiently, [3] proposed the following
  – Assume message depth tried first is $a$, followed by $b$, then finally $c$ if needed
  – Nodes that receive search at depth $a$ store the search temporarily
  – If originating node is satisfied, it doesn’t resend, nodes at depth $a$ will delete the message after a time
Iterative Deepening

– If needed, search is repeated after a delay, TTL is set to $a$.
  • Nodes on the way to $a$ recognize the repeat message and ignore
  • Upon reaching depth $a$, the message is forwarded with TTL of $b - a$
  • Search is saved at level $b$ now
  • Gnutella ID used to track packets
– Finally, if it needs to be repeated again, new packets go out from $b$. All nodes on the way to depth $b$ ignore. Since $c$ is final level, no need to save on this third iteration
– This requires knowledge by nodes of the policy
Directed BFS

- Directed BFS works by selecting a subset of the neighbors, and sending to these
- You are trying to guess who is most likely to respond
- That neighbor may pass on using standard BFS, or the neighbor may use Directed BFS to further limit its messages
- Trick is finding a good way to select neighbors
Selection Strategies in BFS

• Random?
• Highest # results for recent queries
• Results w/low hop counts
• High message count
• Short message queue (flow control)
• Returns results quickly
• Has most neighbors
• Shortest latency
Local Indices

• Idea is that a node maintains information about what files neighbors, and possibly neighbor’s neighbors store. “radius of knowledge”

• All nodes know about radius, and know about policy
  – Policy lists which levels will respond and which will ignore messages
  – Servents look at TTL/Hops to determine if they process or ignore
Local Indices

• Memory issue of maintaining lists
  – Size is far below a megabyte for radius < 5

• Network issue of building list
  – Hit from extra packets
Random Walkers

• Quite a different approach
  – Doesn't use TTLs in same way
• One or a few messages are sent out a random link, if the object is found, return a hit, otherwise send to a different node
• Choice of where to go next is essentially random
• Periodically, when a node receives a query, it checks with source to see if query has been satisfied
Random Walks

• Basic random walk (one walker)
  – Choose a random neighbor at each step
  – “Walker” carries the query to search for each hop
Random Walks (cont.)

- Advantage: cut message overhead (av. #msg. per node)
- Disadvantage: increase query searching delay(#hops)
Multiple Parallel Random Walks (K Walker)

• To decrease delay, increase walkers
• Multiple Walkers
  – K walkers, T steps = 1 walker, k*T steps therefore cut the delay by factor of k.
How to terminate the walkers?

- TTL? No
  - Terminate after certain number of hops
  - Runs into the TTL selection issue too

- checking? YES
  - Walker checks with request node frequently
  - Checking once at every fourth step along the way is usually good
One improvement: State Keeping

• Each query has a unique ID and All its K walkers are tagged with that ID
• For each ID, a node only forwards the query to a neighbor who is new to this query
Issues

• Several alternatives (Local Indices, Iterative Deepening) require a global policy to be understood by all nodes
• Sharing information about file index (Local Indices) or even statistics (Directed BFS) leads to possible security risks
• Most, require at least some modification to the servants
Evaluating Techniques

• Do the results suffer?
  – Most of these assume a request to be satisfied with a number of hits – typically 50-100.
  – Time for those results?
• What sort of network traffic does this add?
• What sort of processing is needed?
• What sort of memory is needed?
• Local state and increased client complexity?
Iterative Deepening

• Savings depend on how long to wait between iterations, depth of each search
  – 80% bandwidth, 60% processing for wait of 8 seconds between iterations
• A policy with depths of 5, 6, 7 and 6 second delays optimal
  – Saves 70% bandwidth, 50% processing
  – Adds only ~30% to time to satisfy
• Some state needed for knowing policy, storing packets
Directed BFS

• Using neighbors with greatest number of results and lowest avg. response time worked best
  – Past performance is an indicator of future performance
• Both had probability of satisfying >85% of a BFS
• About 70% less bandwidth and processing
• Higher cost than iterative deepening, but slightly faster
Local Indices

- Same results claimed as BFS
- No timing results, since simulated on existing data
- Traffic is reduced by 60%, processing by 50% for a radius of 1
- Index size is a factor – radius of 1 71KB, but 7 is 21MB!
- Bandwidth actually increases for large $r$ – too much traffic to maintain indices
Random Walkers

- Can have large gain in efficiency if
  - State to prevent duplicates
  - Good policy to terminate once satisfied (not TTL based)
  - Very good results obtained for low number of hops (as low as 8, which is similar to depth used for floods)
  - Looks like a promising area of research, perhaps combined with some metrics used in Directed BFS to select nodes
Conclusion

• Almost all improve, but trade-offs
  – Most need to maintain state, increased coding complexity
  – Usually less results, slower performance
  – Security implications of more information sharing

• Good deal of research left to be done in this area, although most work appears to be a year or more old
References


