
This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.
Introduction

Definition
A peer-to-peer system is a collection of peer nodes, that act both as servers and as clients
- Provide resources to other peers
- Consume resources from other peers

Characteristics
- Put together resources at the edge of the Internet
- Share resources by direct exchange between nodes
- Perform critical functions in a decentralized manner
Motivation for P2P

- **Cost-effective**
  - Exploit the “dark matter” of the Internet constituted by “edge” resources

- **No central point of failure**
  - Control and resources are decentralized

- **Scalability**
  - Since every peer is alike, it is possible to add more peers to the system and scale to larger networks
It’s a broad area...

- **P2P file sharing**
  - Gnutella
  - eMule
  - BitTorrent

- **P2P communication**
  - Instant messaging
  - Voice-over-IP: Skype

- **P2P computation**
  - Seti@home

- **DHTs & their apps**
  - Chord, CAN, Kademlia, ...

- **P2P wireless**
  - Ad-hoc networking
Overlay networks
Overlay networks

Virtual edge
- TCP connection
- or simply a pointer to an IP address

Overlay maintenance
- Periodically ping to make sure neighbor is still alive
- Or verify liveness while messaging
- If neighbor goes down, may want to establish new edge
- New node needs to bootstrap
Overlay networks

Tremendous design flexibility

- Topology
- Message types
- Protocols
- Messaging over TCP or UDP

Underlying physical net is transparent to developer

- But some overlays exploit proximity
Overlay Topology

Unstructured:
- No explicit topology
- Observed rather than engineered
- Example: Gnutella, BitTorrent

Structured:
- An explicit “shape” is maintained
- Examples: Rings, Trees, DHTs
- Random topologies are “structured” as well

Centralized

Hierarchical

Decentralized

Hybrid
Criteria for topology selection

- Does it simplify location of data?
- Does it
  - balance the load, if nodes are equal?
  - exploit heterogeneity, otherwise?
- Is it robust?
  - Can it work if part of it is suddenly removed?
  - Can it be maintained in spite of churn?
- Has some correspondence with the underlying network topology?
  - Proximity (latency-based)
  - e.g., Pastry, Kazaa, Skype
Distributed Hash Table (DHT)

A peer-to-peer algorithm that offers an associative Map interface:

- **put**(Key $k$, Value $v$): associate a value $v$ to the key $k$
- **Value get**(Key $k$): returns the value associated to key $k$

(Distributed) Hash Tables:

- Hash tables map keys to memory locations
- Distributed hash tables map keys to nodes

Organization:

- Each node is responsible for a portion of the key space
- Messages are routed between nodes to reach responsible nodes
- Replication used to tolerate failures
Routing in DHTs

1. `put(9, "x")`
2. `get(9)`
3. Routing path in the DHT network.
DHT Implementations

- The founders (2001):
  - Chord
  - CAN
  - Pastry
  - Tapestry

- The ones which are actually used:
  - Kademlia and its derivatives (up to 4M nodes!)
    - BitTorrent
    - Kad (eMule)
    - The Storm Botnet
  - Cassandra DHT
    - Part of Apache Cassandra
    - Initially developed at Facebook

- The ones which are actually used, but we don’t know much about:
  - Microsoft DHT based on Pastry
  - Amazon’s Dynamo key-value store
Step 1: From Keys and Nodes to IDs

- Keys and nodes are represented by identifiers taken from an ID space
  - Key identifiers: computed through a hash function (e.g., SHA-1)
    - e.g., \( ID(k) = SHA1(k) \)
  - Node identifiers: randomly assigned or computed through a hash function
    - e.g., \( ID(n) = SHA1(\text{IP address of } n) \)

Why?

- Very low probability that two nodes have exactly the same ID
- Nodes and keys are mapped in the same space
Step 2: Partition the ID space

- Each node in the DHT stores some $k,v$ pairs
- Partition the ID space in zones, depending on the node IDs:
  - A pair $(k,v)$ is stored at the node $n$ such that (examples):
    - its identifier $ID(n)$ is the closest to $ID(k)$;
    - its identifier $ID(n)$ is the largest node id smaller than $ID(k)$
Step 2: Build overlay network

Each node has two sets of neighbors:

- Immediate neighbors in the key space (leafs)
  - Guarantee correctness, avoid partitions
  - If we had only them, linear routing time

- Long-range neighbors
  - Allow sub-linear routing
  - If we had only them, connectivity problems
Step 3: Route puts/gets through the overlay

- **Recursive routing**: the initiator starts the process, contacted nodes forward the message
- **Iterative routing**: the initiator personally contact the nodes at each routing step

---

### Recursive routing

0 \[\rightarrow\] 2 \[\rightarrow\] 3 \[\rightarrow\] 2 \[\rightarrow\] 1 \[\rightarrow\] get

---

### Iterative routing

0 \[\rightarrow\] 1 \[\rightarrow\] 5 \[\rightarrow\] 6 \[\rightarrow\] 4 \[\rightarrow\] 3 \[\rightarrow\] 2 \[\rightarrow\] get
Routing around failures (1)

- Under churn, neighbors may have failed
- To detect failures, acknowledge each hop (recursive routing)
Routing around failures (2)

- If we don’t receive ack or response, resend through a different neighbor

Recursive routing

Iterative routing
Routing around failures (3)

- Must compute timeouts carefully
  - If too long, increase put/get latency
  - If too short, get message explosion

- Parallel sending could be a design decision – see Kademlia
Computing good timeouts

- Use TCP-style timers
  - Keep past history of latencies
  - Use this to compute timeouts for new requests

- Works fine for recursive lookups
  - Only talk to neighbors, so history small, current

- In iterative lookups, source leads the entire lookup process
  - Must potentially have good timeout for any node
Recovering from failures

- Can’t route around failures forever
  - Will eventually run out of neighbors

- Must also find new nodes as they join
  - Especially important if they’re our immediate predecessors or successors
Recovery from failures

- **Reactive recovery**
  - When a node stops sending acknowledgments, notify other neighbors of potential replacements

- **Proactive recovery**
  - Periodically, each node sends its neighbor list to each of its neighbors
Chord

- ID space: uni-dimensional ring in $[0, 2^m - 1]$ ($m = 160$)
- Routing table size: $O(\log n)$
- Routing time: $O(\log n)$

Bibliography


Identifier mapping

Example:
- Node 8 maps [5, 8]
- Node 15 maps [9, 15]
- Node 20 maps [16, 20]
- ... 
- Node 4 maps [59, 4]

Random ID assignment
- Each node maintains a pointer to its successor
Join procedure (1)

Example:

- Node with $id = 50$ joins the ring
- Node 50 needs to know at least one node already in the system
- Assume known node is 15
Join procedure (2)

Example:

- Node 50: send \langle JOIN, 50 \rangle to node 15
- Message is routed to node 44
- Node 44: returns node 58
- Node 50: updates its successor to 58
Stabilization

- Periodically, each node $A$:
  - sends a $\langle\text{STABILIZE}\rangle$ message to its successor $B$

- Upon receiving $\langle\text{STABILIZE}\rangle$ message from $A$, node $B$:
  - returns its predecessor $B' = \text{pred}(B)$ to $A$ by sending a $\langle\text{NOTIFY}, B'\rangle$ message
  - updates its predecessor to $A$, if $A$ is between $B'$ and $B$

- Upon receiving $\langle\text{NOTIFY}, B'\rangle$ message from $B$, node $A$:
  - updates its successor to $B'$, if $B'$ is between $A$ and $B$
Join procedure (4)

Example:

- Node 50: send \texttt{〈STABILIZE〉} to node 58
- Node 58: update predecessor to 50
- Node 58: send \texttt{〈NOTIFY, 50〉} back
Example:

- Node 44: send \(\langle\text{STABILIZE}\rangle\) to its successor node 58
- Node 58: replies with \(\langle\text{NOTIFY}, 50\rangle\)
- Node 44: updates its successor to 50
Join procedure (6)

Example:
- Node 44: send \langle \text{STABILIZE} \rangle to its new successor, node 50
- Node 50: updates its predecessor to 44

This completes the joining operation!
Achieving efficiency

- Chord requires each node to keep a **finger table** containing up to \( m \) entries
- The \( i \)-th entry \((0 \leq i \leq m - 1)\) of node \( n \) will contain the address of the successor of \((n + 2^i) \mod 2^m\)
- Fingers are used in routing to reduce the number of hops to \( O(\log N) \)
Achieving efficiency

\[
(80 + 2^6) \mod 2^7 = 16
\]

<table>
<thead>
<tr>
<th>i</th>
<th>ft[i]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>96</td>
</tr>
<tr>
<td>1</td>
<td>96</td>
</tr>
<tr>
<td>2</td>
<td>96</td>
</tr>
<tr>
<td>3</td>
<td>96</td>
</tr>
<tr>
<td>4</td>
<td>96</td>
</tr>
<tr>
<td>5</td>
<td>112</td>
</tr>
<tr>
<td>6</td>
<td>20</td>
</tr>
</tbody>
</table>
Achieving robustness

- To improve robustness, each node maintains $k > 1$ immediate successors instead of only one.
- In the ⟨NOTIFY⟩ message, node $A$ can send its $k - 1$ successors to its predecessor $B$.
- Upon receiving the ⟨NOTIFY⟩ message, $B$ can update its successor list by concatenating the successor list received from $A$ with $A$ itself.
Optimizations

- **Reduce latency**
  - Choose finger that reduces expected time to reach destination
  - Choose the closest node from range \([n + 2^{i-1}, n + 2^i]\) as successor

- **Accommodate heterogeneous systems**
  - Multiple virtual nodes per physical node
Associate to each node and item a unique ID in an $d$-dimensional Cartesian space on a $d$-torus

Routing table size is constant: $O(d)$

Guarantees that a key is found in at most $d \cdot n^{1/d}$ steps, where $n$ is the total number of nodes

Bibliography

In Proc. of SIGCOMM'01, pages 161–172, San Diego, California, USA, 2001. ACM.
http://www.disi.unitn.it/~montreso/ds/papers/CAN.pdf
Example: 2-dimensional space

- Space divided between nodes
- All nodes cover the entire space
- Each node covers either a square or a rectangular area of ratios 1 : 2 or 2 : 1

Example:
- Node $n_1 : (1, 2)$ – first node that joins – cover the entire space
Example: 2-dimensional space

- Node $n_2 : (4, 2)$ joins: space is divided between $n_1$ and $n_2$
Example: 2-dimensional space

- Node $n_3 : (3, 5)$ joins
Example: 2-dimensional space

Example:

- Nodes $n_4 : (5, 5)$ and $n_5 : (6, 6)$
  - join
Example: 2-dimensional space

Example:

- Nodes: \( n_1 : (1, 2), n_2 : (4, 2), n_3 : (3, 5), n_4 : (5, 5), n_5 : (6, 6) \)
- Items: \( k_1 : (2, 3), k_2 : (5, 1), k_3 : (2, 1), k_4 : (7, 5) \)
- Each item is stored by the node who owns its mapping in the space
Example: 2-dimensional space

Example:
- Each node knows its neighbors in the $d$-space
- Forward query to the neighbor that is closest to the query id
- Example: assume $n_1$ queries $k_4$
- Can route around some failures
Example: 2-dimensional space

Node joining:

1. Discover some node $I$ already in CAN
2. Pick random point $(x, y)$ in space
3. $I$ routes to $(x, y)$, discovers node $J$
Example: 2-dimensional space

Node joining:

1. Split $J$ zone in half
2. New node owns one half
Node departures

Take-over mechanism:

- Node explicitly hands over its zone and the associated (key,value) database to one of its neighbors
- A maximum of $2d$ nodes need to be contacted
- Problem: in case of network failure, no regeneration of data
- Solution: every node has a backup of its neighbors
Multi-verse?

Increasing availability:

- Each key is mapped into $r$ different realities
- Each reality is associated with a different hash function
- A key is not available only when the $r$ nodes hosting it in different realities are down at the same time
Kademlia

Key points

- Kademlia uses tree-based routing
- SHA-1 hash function in a 160-bit address space
- Every node maintains information about keys close to itself
  - Distance based on the XOR metric: $d(a, b) = a \oplus b$
- Uses parallel asynchronous queries to avoid timeout delays
- Routes are selected based on latency

Bibliography

P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the XOR metric.
Kademlia Tree

- Nodes are treated as leaves in binary tree
- Node’s position in the tree is determined by the shortest unique prefix of its ID
- A node is responsible for all “closest” IDs (those having same prefix as itself)
From the point of view of each node, the tree is divided into a series of maximal subtrees that do not contain the node.

Example: the red node with prefix 0011

A node must know at least one node in each of these subtrees.
Routing table

Consider routing table for a node with prefix 0011
The routing table is composed of a series of $k$-buckets corresponding to each of the subtrees
Consider a 2-bucket example, each bucket will have at least 2 contacts for each subtree
Consider a query for ID 111010... initiated by node 0011100...
Kademlia protocol consists of 4 RPCs:

- $\text{ping}_{n \rightarrow m}()$
  - Probe node $m$ to see if it is online

- $\text{store}_{n \rightarrow m}(k, v)$
  - Instruct node $m$ to store a $\langle k, m \rangle$ pair

- $\text{findNode}_{n \rightarrow m}(t)$
  - Returns the $k$ contacts “closest” to $t$

- $\text{findValue}_{n \rightarrow m}(k)$
  - Returns the value associated to $k$, if present, or
  - Returns $k$ contacts closest to $k$
Routing

Goal: find \( k \) nodes closest to ID \( t \) – Protocol executed by \( n_0 \)

- **Initial phase**:
  - insert in a set \( S \) all the nodes in the routing table

- **Iteration**
  - select a subset \( T \subseteq S \) of the \( \alpha \) nodes closest to \( t \)
  - invoke \( \text{findNode}(t) \) on nodes in \( T \), in parallel
  - collect the replies in a new set \( S \)
  - repeat until no new node is discovered

- **Final phase**
  - invoke \( \text{findNode}(t) \) to all of \( k \) closest nodes not already queried
  - return when have results from all the \( k \)-closest nodes
Kademlia summary

Strengths:

- Low control message overhead
- Tolerance to node failure and leave
- Capable of selecting low-latency path for query routing
- Unlike Chord, Kademlia is symmetric: $a \oplus b = b \oplus a$
  - Peers receive lookup queries from precisely the same set of neighbors contained in their routing tables

Weaknesses:

- Balancing of storage load is not truly solved
- No experimental results provided
Cassandra

Few information available:

- $O(1)$ routing hops
- $O(N)$ routing state
  - Thanks to a routing protocol that guarantees that eventually every node knows every other node

Bibliography

D. Featherston. Cassandra: Principles and application.
Security aspects of DHTs

Security weaknesses specific to DHTs

- **Sybil attacks**
  - an attacker introduces a large number of bogus nodes that can subvert protocols based on redundancy

- **Eclipse attacks**
  - an attacker tries to corrupt the routing tables of honest nodes by filling them with references to malicious nodes

- **Routing and storage attacks**
  - various attacks where malicious nodes do not follow the routing and storage protocols correctly

Bibliography


Example of attacks

Any routing decision or storage manipulation possible

Key of entity

Sybil attacker with multiple IDs

Eclipsed node pointing to malicious peers

Routing table

Actual node

Reference to malicious node

Issues that are inherent to all DHT deployments, but which are independent of the associated protocols, such as churn and unbalanced loads, as well as application-specific attacks, are out of the scope of this paper. In the same light, we do not discuss denial of service attacks, which have been studied by Daswani [2004].

There have been several surveys that describe DHTs and peer-to-peer (P2P) systems in general. However, not many survey solutions to security issues in DHTs. Sit and Morris [2002] explore the subject and provide general guidelines. Castro et al. [2002] study DHT security issues under a generic DHT model, and provide solutions using Pastry as a representative of their model. Wallach [2002] discusses a wide range of security issues in several P2P systems, including Pastry, but does not enumerate the numerous proposals. Srivatsa and Liu [2004] make an extensive quantitative analysis of security threats in DHTs and some of the defenses. Levine et al. [2006] summarize general approaches to address the Sybil attack in a variety of scenarios, but do not discuss any specific measures. Reidemeister et al. [2005] study security issues specific to CAN [Ratnasamy et al. 2001]. Dahan and Sato [2007] criticize several practical aspects related to DHT security as well as their use in other systems that require security.

In this paper, we supplement these surveys by providing a comprehensive overview of the research in the area of DHT security, concentrating on numerous specific solutions. We focus on proposed defenses against the aforementioned attacks, discuss their advantages, and analyze their effectiveness.
Defenses against Sybil attacks

- Collusion is easier

- Possible defenses:
  - Centralized certification
  - Distributed registration
  - Physical network characteristics
  - Social networks
  - Computational puzzles

- You can only reduce the impact of Sybil attacks, not eliminate them completely
Defenses against eclipse attacks

- Effect of eclipse attack ("table poisoning") is measured by:
  \[
  \frac{\text{percentage of malicious entries in routing tables}}{\text{percentage of malicious users in the network}}
  \]

- Possible defenses:
  - Constrained neighbor selection
    - Original Chord: only one node may fit in a finger table entry – good
    - Random Chord: several nodes may fit in finger table entry – bad
    - Pastry: some table entries may be filled by any node sharing a short prefix – bad
    - Kademlia: table entries are filled by fast-responding peers – good
  - In-degree anonymous auditing
    - Malicious nodes have larger in-degree
Defenses against routing and storage attacks

- Redundant routing
  - Possible approaches:
    - Multiple paths
    - Wide paths
    - Multiple wide paths
  - Wide paths require one good node per hop, multiple paths require a path with only good nodes

- Redundant storage
  - Storing replicas “numerically close” to each other
    - Chord, Pastry, Kademlia
    - Pros: easier to maintain consistency
    - Cons: malicious node may control a region of space
  - Storing replicas spread over the identifier space
    - Tapestry, several other proposals
    - Pros: most difficult to subvert an area
    - Cons: requires additional tables
Why Kademlia?

Generic reasons

- Relative security: wide searches
- Replicated storage

The reality is that Kademlia is insecure

- Successful (academic) attacks on Kad/BitTorrent
- Successful infiltrations on the Storm BotNet

The real reasons

- For BitTorrent, damage is limited anyway (decentralized tracking)
- Many alternative ways to obtain peers (PEX, multiple trackers)
## Comparison

<table>
<thead>
<tr>
<th>Architecture</th>
<th>CAN</th>
<th>Chord</th>
<th>Tapestry</th>
<th>Pastry</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$d$-dimens. space</td>
<td>ring</td>
<td>Plaxton tree</td>
<td>Plaxton tree</td>
</tr>
<tr>
<td>Routing hops</td>
<td>$O(dN^{1/d})$</td>
<td>$O(\log N)$</td>
<td>$O(\log_b N)$</td>
<td>$O(\log_b N)$</td>
</tr>
<tr>
<td>Routing state</td>
<td>$2d$</td>
<td>$\log N$</td>
<td>$\log_b N$</td>
<td>$B \log_b N$</td>
</tr>
<tr>
<td>Join cost</td>
<td>$2d$</td>
<td>$(\log N)^2$</td>
<td>$\log_b N$</td>
<td>$\log_b N$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Kademlia</th>
<th>Viceroy</th>
<th>Koorde</th>
<th>Kelips</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Tree</td>
<td>Butterfly network</td>
<td>de Bruijn graph</td>
<td>$n$-dimens. space</td>
</tr>
<tr>
<td>Routing hops</td>
<td>$O(\log N)$</td>
<td>$O(\log n)$</td>
<td>$O\left(\frac{\log n}{\log \log n}\right)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Routing state</td>
<td>$k \log N$</td>
<td>$\log N$</td>
<td>$\log N$</td>
<td>$\sqrt{n}$</td>
</tr>
<tr>
<td>Join cost</td>
<td>$k \log N$</td>
<td>$\log N$</td>
<td>$\log N$</td>
<td>$\sqrt{n}$</td>
</tr>
</tbody>
</table>
Conclusions

- The DHT abstraction is doing well, both inside clouds and in P2P networks

- Kademlia seems to be the winner. Main reasons:
  - Performance
  - Relative security
Gnutella: brief history

- Nullsoft (a subsidiary of AOL) released Gnutella on March 14th, 2000, announcing it on Slashdot
- AOL removed Gnutella from Nullsoft servers on March 15th, 2000
- After a few days, the Gnutella protocol was reverse-engineered
- Napster was shutdown in early 2001, spurring the popularity of Gnutella
- On October 2010, LimeWire (a popular client) was shutdown by court’s order
Gnutella is a protocol for peer-to-peer search, consisting of:

- A set of message formats
  - 5 basic message types
- A set of rules governing the exchange of messages
  - Broadcast
  - Back-propagate
  - Handshaking
- An hostcache for node bootstrap
Gnutella topology: unstructured

No central authority
Each node selects its own neighbors
Gnutella routing
Gnutella routing
Gnutella routing
Gnutella routing
Gnutella messages

Each message is composed of:

- A 16-byte ID field uniquely identifying the message
  - randomly generated
  - not related to the address of the requester (anonymity)
  - used to detect duplicates and route back-propagate messages

- A message type field
  - PING, PONG
  - QUERY, QUERYHIT
  - PUSH (for firewalls)

- A Time-To-Live (TTL) Field

- Payload length
Gnutella messages

- **PING** (broadcast)
  - Used to maintain information about the nodes currently in the network
  - Originally, a “who’s there” flooding message
  - A peer receiving a PING is expected to respond with a PONG message

- **PONG** (back-propagate)
  - A PING message has the same ID of the corresponding PING message
  - Contains:
    - address of connected Gnutella peer
    - total size and total number of files shared by this peer
Gnutella messages

- **QUERY** (broadcast)
  - The primary mechanism for searching the distributed network
  - Contains the query string
  - A servent is expected to respond with a QUERYHIT message if a match is found against its local data set

- **QUERYHIT** (back-propagate)
  - The response to a query
  - Has the same ID of the corresponding QUERY message
  - Contains enough information to acquire the data matching the corresponding query
    - IP Address + port number
    - List of file names
Beyond the original Gnutella

Several problems in Gnutella 0.4 (the original one):

- What kind of topology is generated?
  - Is it planned (“engineered”)?
  - Is it good?

- PING-PONG traffic
  - More than 50% of the traffic generated by Gnutella 0.4 is PING-PONG related

- Scalability
  - Each query generates a huge amount of traffic
    - e.g. $TTL = 6, d = 10 \Rightarrow 10^6$ messages
    - Potentially, each query is received multiple times from all neighbors
Gnutella overlay vs underlying topology

Unfortunately, it is prohibitively expensive to compute exactly the mapping of the Gnutella onto the Internet topology, due both to the inherent difficulty of extracting Internet topology and to the computational scale of the problem. Instead, we proceed with two high-level experiments that highlight the mismatch between the topologies of the two networks.

The Internet is a collection of Autonomous Systems (AS) connected by routers. ASs, in turn, are collections of local area networks under a single technical administration. From an ISP point of view traffic crossing AS borders is more expensive than local traffic. We found that only 2-5% of Gnutella connections link nodes located within the same AS, although more than 40% of these nodes are located within the top ten ASs. This result indicates that most Gnutella-generated traffic crosses AS borders, thus increasing costs, unnecessarily.

In the second experiment we assume that the hierarchical organization of domain names mirrors that of the Internet infrastructure. For example, it is likely that communication costs between two hosts in the “uchicago.edu” domain are significantly smaller than between “uchicago.edu” and “sdsc.edu.” The underlying assumption here is that domain names express some sort of organizational hierarchy and that organizations tend to build networks that exploit locality within that hierarchy.

In order to study how well the Gnutella virtual topology maps on to the Internet partitioning as defined by domain names, we divide the Gnutella virtual topology graph into clusters, i.e., subgraphs with high interior connectivity. Given the flooding-like routing algorithm used by Gnutella, it is within these clusters that most load is generated. We are therefore interested to see how well these clusters map on the partitioning defined by the domain naming scheme.

We use a simple clustering algorithm based on the connectivity distribution described earlier: we define as clusters subgraphs formed by one hub with its adjacent nodes. If two clusters have more than 25% nodes in common, we merge them. After the clustering is done, we (1) assign nodes that are included in more than one cluster only to the largest cluster and (2) form a last cluster with nodes that are not included in any other cluster.

We define the entropy [11] of a set $C$, containing $|C|$ hosts, each labeled with one of the $n$ distinct domain names, as:

$$H_C = - \sum_{i=1}^{n} p_i \log(p_i) \quad \text{bits}$$

where $p_i$ is the probability of randomly picking a host with domain name $i$. 

The figure shows two different mappings of Gnutella's virtual network topology (blue, dotted arrows) to the underlying network infrastructure (black, solid lines). The left picture: perfect mapping. A message inserted into the network by node A travels physical link D-E only once to reach all other nodes. The right picture: inefficient mapping. The same distribution requires that the message traverse physical link D-E six times.
When analyzing global connectivity and reliability patterns in the Gnutella network, it is important to keep in mind the self-organized network behavior: users decide only the maximum number of connections a node should support, and nodes decide whom to connect to or when to drop/ad a connection based only on local information.

Recent research [1,7,8,13] shows that many natural networks such as molecules in a cell, species in an ecosystem, and people in a social group organize themselves as so-called power-law networks. In these networks most nodes have few links and a tiny number of hubs have a large number of links. More specifically, in a power-law network the fraction of nodes with L links is proportional to $kL^{-\alpha}$, where $k$ is a network dependent constant.

This structure helps explain why networks ranging from metabolisms to ecosystems to the Internet are generally highly stable and resilient, yet prone to occasional catastrophic collapse [14]. Since most nodes (molecules, Internet routers, Gnutella servers) are sparsely connected, little depends on them: a large fraction can be taken away and the network stays connected. But, if just a few highly connected nodes are eliminated, the whole system could crash. One implication is that these networks are extremely robust when facing random node failures, but vulnerable to well-planned attacks.

Given the diversity of networks that exhibit power-law structure and their properties we were interested to determine whether Gnutella falls into the same category. Figure 5 presents the connectivity distribution in Nov. 2000. Although data are noisy (due to the small size of the networks), we can easily recognize the signature of a power-law distribution: the connectivity distribution appears as a line on a log-log plot. [6,4] confirm that early Gnutella networks were power-law. Later measurements (Figure 6) however, show that more recent networks tend to move away from this organization: there are too few nodes with low connectivity to form a pure power-law network. In these networks the power-law distribution is preserved for nodes with more than 10 links while nodes with fewer links follow an almost constant distribution.

Figure 5: Connectivity distribution during November 2000. Each series of points represents one Gnutella network topology we discovered at different times during that month. Note the log scale on both axes. Gnutella nodes organized themselves into a power-law network.

Figure 6: Connectivity distributions during March 2001. Each series of points represents one Gnutella network topology discovered during March 2001. Note the log scale on both axes. Networks crawled during May/June 2001 show a similar pattern.

We speculate there are two reasons for the peculiar distribution in Figure 6. First, Gnutella users are technically savvy users, early technology adopters. The percentage of Gnutella users with modem connection is significantly lower than in the Internet users population: less than 20% users connect...
Connectivity (and robustness)

Figure 1: Gnutella network growth. The plot presents the number of nodes in the largest connected component in the network. Data collected during Nov. 2000, Feb./March 2001, and May 2001. We found a significantly larger network around Memorial Day (May 24-28) and Thanksgiving 2000, when apparently more people hunt for shared music online.

Figure 2: Generated traffic (messages/sec) in Nov. 2000 classified by message type over a 376 minute period. Note that overhead traffic (PING messages, that serve only to maintain network connectivity) formed more than 50% of the traffic. The only 'true' user traffic is QUERY messages. Overhead traffic has decreased by May 2001 to less than 10% of all generated traffic.

Using records of successive crawls, we investigate the dynamic graph structure over time. We discover that about 40% of the nodes leave the network in less than 4 hours, while only 25% of the nodes are alive for more than 24 hours. Given this dynamic behavior, it is important to find the appropriate tradeoff between discovery time and invasiveness of our crawler. Increasing the number of parallel crawling tasks reduces discovery time but increases the burden on the application. Obviously, the Gnutella map our crawler produces is not an exact 'snapshot' of the network. However, we argue that the network graph we obtain is close to a snapshot in a statistical sense: all properties of the network: size, diameter, average connectivity, and connectivity distribution are preserved.
Gnutella conclusions

Gnutella 0.6:
- Superpeer-based organization
- Ping/pong caching
- Query routing

Summary:
- A milestone in P2P computing
  - Gnutella proved that full decentralization is possible
- But:
  - Gnutella is a patchwork of hacks
  - The ping-pong mechanism, even with caching, is just plain inefficient
Interest on P2P system driven by file sharing applications
  ▶ end users become content provider

Main focus is to efficiently discover content
  ▶ different generations of P2P...
    ★ centralized (Napster), unstructured (Gnutella), structured (DHT)
  ▶ ...with different problems
    ★ single point of failure (centralized), low success rate (unstructured),
      high management traffic (structured)

But...what happens when you find the content?
BitTorrent

- Designed for efficient content download
- Search features not included
- Large portion of the Internet traffic is due to BitTorrent
- Basic concept: file swarming

Bibliography


Legal (!) applications

- Music, video and the like
  - BitTorrent Inc
  - SubPop Records
  - Norwegian Broadcasting Corporation

- Software
  - Linux distributions
  - Blizzard: Diablo III, StarCraft II, World of Warcraft (game updates)

- Web services
  - Amazon S3 equipped with built-in BitTorrent support
  - Facebook, Twitter use BitTorrent to distribute updates to their servers
BitTorrent architecture

1. file.torrent

web server

tracker

2. random peer set

file.xvid

P1 P2 P3

leechers

seeders
A torrent file is a \textit{bencoded} dictionary with the following keys:

- \textbf{announce} – the URL of the tracker
- \textbf{name} – suggested file/directory name
- \textbf{piece length} – number of bytes per piece (commonly 256KB)
- \textbf{pieces} – a concatenation of each piece’s SHA-1 hash.

Exactly one of \textbf{length} or \textbf{files}:

- \textbf{length} – size of the file (in bytes)
- \textbf{files} – a list of files with the following keys:
  - \textbf{path} - pathname of the file
  - \textbf{length} - size of the file (in bytes)
BitTorrent architecture

**Peer Selection**
- “Which peers to upload to”
- Efficiency criteria:
  - ★ Maximize service capacity
  - ★ Foster reciprocation and prevent free riders

**Piece selection**
- “Which pieces to download from selected peer”
- Should guarantee *piece diversity*
  - ★ Always find an interesting piece in selected peer
  - ★ Do not bias peer selection
Piece selection

- The order in which pieces are selected by peers is critical
- A bad algorithm could create a situation where all peers have all pieces that are currently available and none of the missing ones
- If the original seed disappears, the download cannot be completed!
Policies

- **Strict Priority**
  - A piece is broken into sub-pieces (typically 16KB in size)
  - Policy: *Until a piece is assembled, only download sub-pieces for that piece from the same source*
  - This policy lets complete pieces assemble quickly

- **Rarest first**
  - Policy: *Determine the pieces that are most rare among your peers and download those first*
  - This ensures that the most common pieces are left till the end to download
  - Rarest first also ensures that a large variety of pieces are downloaded from the seed
Policies

- **Random first piece**
  - Initially, a peer has nothing to trade
  - Important to get a complete piece ASAP
  - Rare pieces are typically available at fewer peers, so downloading a rare piece initially is not a good idea
  - Policy: *Select a random piece of the file and download it*

- **Endgame mode**
  - Policy: *When all the sub-pieces that a peer doesn’t have are actively being requested, these are requested from every peer*
  - When the sub-piece arrives, the replicated requests are canceled
  - This ensures that a download doesn’t get prevented from completion due to a single peer with a slow transfer rate
  - Some bandwidth is wasted; in practice, not too much
Peer selection

Choking

- Choking is a temporary refusal to upload; download occurs as normal
- One of BitTorrent’s most powerful idea
- It ensures that nodes cooperate and eliminates (?) the free-ride problem
- When a node is unchoked, upload restart
- Connection is kept open to reduce setup costs
- Based on game-theoretic tit-for-tat strategy in repeated games
## Prisoner’s Dilemma

Two men are arrested, but the police do not possess enough information for a conviction. Following the separation of the two men, the police offer both a similar deal:

<table>
<thead>
<tr>
<th></th>
<th>Prisoner $B$ stays silent</th>
<th>Prisoner $B$ confesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prisoner $A$ stays silent</td>
<td>Both serve 1 months</td>
<td>$A$ serves 1 year</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$B$ goes free</td>
</tr>
<tr>
<td>Prisoner $A$ confesses</td>
<td>$B$ serves 1 year</td>
<td>Both serve 3 months</td>
</tr>
<tr>
<td></td>
<td>$A$ goes free</td>
<td></td>
</tr>
</tbody>
</table>
Prisoner’s Dilemma

Single-iteration game

- What is the best strategy?
- “Confessing” is a dominant strategy
  - If the other prisoner confesses, the best move is to confess
  - If the other prisoner stay silent, the best move is to confess

What about iterated games?

- Robert Axelrod’s “The evolution of cooperation”
- Tournament of computer programs playing PD
- The winner: Tit-for-tat, Anatol Rapoport
Prisoner’s Dilemma

**Tit-for-tat**
- Be nice at the beginning
- Do onto others as they do onto you:
- If the other prisoner confesses, you must retaliate back
- Have a recovery mechanism to ensure eventual cooperation

How to translate this in BitTorrent?
Choking/unchoking

**Goal**: have several bidirectional connections running continuously

- Upload to peers who have uploaded to you recently
  - “Do onto others as they do onto you”

- Unused connections are uploaded to on a trial basis to see if better transfer rates could be found using them
  - “Be nice at the beginning”
  - “Have a recovery mechanism to ensure eventual cooperation”
Choking/unchoking specifics

- A peer always unchokes a fixed number of its peers (default: 4)
- Decision to choke/unchoke done based on current download rates, averaged over the last 20s
- Evaluation on who to choke/unchoke is performed every 10s
  - Prevents wasting of resources by rapidly choking/unchoking peers
  - Enough for TCP to ramp up transfers to their full capacity
- Which peer is the optimistic unchoke is rotated every 30s
  - Used to discover if a currently choked peer would be better
Additional details

Anti-snubbing:

- A peer is said to be *snubbed* if each of its peers chokes it.
- To handle this, snubbed peer stops uploading to its peers.
- Optimistic unchoking done more often:
  - Hope is that will discover a new peer that will upload to us.

Seeding:

- Once download is complete, a peer has no download rates to use for comparison nor has any need to use them.
- The question is, which nodes to upload to?
- Policy: Upload to those with the best upload rate.
  - This ensures that pieces get replicated faster.
Improvements over the tracker bottleneck

- **Trackerless BitTorrent** (i.e., w/o a centralized tracker):
  - Based on variants of Kademlia DHT
  - Tracker run by a normal end-host
  - Vuze DHT vs Mainline DHT

- **Peer Exchange (PEX):**
  - Each peer directly update other peers as to which peers are currently in the swarm
  - Epidemic sampling!
  - Three incompatible version of PEX (Vuze, BitComet, Mainline)

- **Multitracking**
  - Multiple trackers in the torrent file
Five months in a torrent’s lifetime

- Analysis of a tracker log
- 1.77GB Linux Redhat 9 distribution
- Five months - April-August 2003
- 180.000 downloads

Bibliography


Dissecting bittorrent: Five months in a torrent’s lifetime.

Network: Number of active peers over time

Figure: Complete trace
Network: Number of active peers over time

**Figure:** First five days
Network: Proportion of seeders and leechers

Figure: Complete trace
Client: Cumulative download and upload evolution

**Figure:** Complete torrent

![Cumulative download and upload evolution graph](image)
Client: Cumulative download and upload evolution

Figure: First ten minutes
Client: Number of connected peers

Figure: Around 14 hours
Cheating BitTorrent

- Tit-for-tat strategy has been designed to foster reciprocation
- Nevertheless, its incentives are not robust to strategic clients
- Two examples:
  - BitTyrant
    - a strategic client that tries to improve download/upload rate
  - BitThief
    - a client that never uploads anything

Bibliography

- M. Piatek, T. Isdal, T. E. Anderson, A. Krishnamurthy, and A. Venkataramani. Do incentives build robustness in BitTorrent?
  In *Proc. of NSDI’07*, Cambridge, Massachusetts, USA, Apr. 2007. USENIX.

- T. Locher, P. Moor, S. Schmid, and R. Wattenhofer. Free riding in BitTorrent is cheap.
BitTyrant

How to improve performance?

- Maximize reciprocation bandwidth per connection
- Maximize number of reciprocating peers
- Deviate from equal split

Unchoking algorithm

- $d_p$: download rate of connection $p$
- $u_p$: upload rate of connection $p$
- Each round, rank peers by the ratio $u_p/d_p$ and unchoke the first $k$ such that the upload capacity is reached:

$$\sum_{i=1}^{k} u_i \leq cap$$
BitTyrant

Figure 11: Download times and sample standard deviation comparing performance of a singleBitTyrant client and an unmodified Azureus client on a synthetic PlanetLab. Further, the consistent performance of BitTyrant in comparison to unmodified Azureus across trials spanning several hours. This experiment did not recover from its initial set of competitive conditions in six months. However, because of the bandwidth capacity of PlanetLab nodes and the distribution of download times, we rely on swarm properties, strategic behavior, and performance.

PlanetLab is often oversubscribed and shares bandwidth dance with our observed distribution. However, because the bandwidth capacity of PlanetLab nodes in accordance with our observations, we rely on swarm properties, strategic behavior, and performance.

We next evaluate using the strategic behavior of users can realize significant performance benefits from observed swarms overall. For most real swarms today, Tyrant mance differed by less than 10%.

Account for the 12 swarms for which download performance is provided to a swarm of newly joined users, this effect may be based on luck with respect to the set of initial peers reconnected immediately. More often than not, this circumstance for which BitTyrant benefits from synthetic churn with constant capacity, each node's upload capacity draws from the distribution as well as unchoke bandwidth, and block size. This was sufficient serving a 5 MB file. We did not change the default seed capacities drawn from our scaled distribution. Three seeds with combined capacity of 128 KB/s were located at UW and an unmodified Azureus client on a synthetic Planet-Lab swarm.

Unstructured systems

BitTorrent
BitThief

Download only: benefits

- no copyright issues (only contributors are sued)
- conserve resources
- spoil the community

Gains from optimistic unchoking:

- Ask for as many clients as possible
  - Increment tracker polling
  - Decentralized tracking, PEX
- Connect to all available clients
  - higher chance of being unchoked
- Always pretend to be a newcomer
  - Advertise no pieces
  - Download whatever available
  - Most clients are nice

Gains from free sharing of seeders:

- Seeders select peers in two ways:
  - highest bandwidth
  - round robin
- BitThief report high upload rate
BitThief

The results are summarized in Figure 2. As a first observation, note that in every experiment, BitThief succeeded in acquiring the entire torrent. More interestingly, we can see that the relative download times vary significantly across the different torrents. The plot shows relative download times with the average download time as the bar. Exceptions are Torrents E and G, where the average download time was significantly lower. Exceptions are Torrents D and E, where the average download time was significantly lower.

In a first experiment, we did not impose any restrictions on our client, in particular, BitThief was also allowed to download from other leechers. Interestingly, as we will see, even without downloading from seeders, BitThief seems to have an advantage over the official client as it had to upload over 3.5GB of data. Torrents A and B are relatively small files, BitThief seems to have an advantage over the official client. Exceptions are Torrents E and G, where the upload was significantly lower. Exceptions are Torrents D and E, where the upload was significantly lower.

In this section, we further constrain BitThief to only downloading from leechers, BitThief can download the whole torrent from leechers. Interestingly, as we will see, even without downloading from seeders, BitThief seems to have an advantage over the official client as it had to upload over 3.5GB of data. Exceptions are Torrents E and G, where the upload was significantly lower. Exceptions are Torrents D and E, where the upload was significantly lower.

The measurements presented so far have all been obtained through experiments on the Internet and hence were subject to various external effects. For example, in case BitThief HotNetsV Session 5: Anti/Social 87, BitThief was, on average, slightly faster than the official client as it had to upload over 3.5GB of data. Torrents A and B are relatively small files, BitThief seems to have an advantage over the official client. Exceptions are Torrents E and G, where the upload was significantly lower. Exceptions are Torrents D and E, where the upload was significantly lower.

Table 1

<table>
<thead>
<tr>
<th>Size</th>
<th>Seeders</th>
<th>Leechers</th>
<th>µ</th>
<th>σ</th>
</tr>
</thead>
<tbody>
<tr>
<td>A 170MB</td>
<td>10518 (303)</td>
<td>7301 (98)</td>
<td>13</td>
<td>4</td>
</tr>
<tr>
<td>B 175MB</td>
<td>923 (96)</td>
<td>257 (65)</td>
<td>14</td>
<td>8</td>
</tr>
<tr>
<td>C 175MB</td>
<td>709 (234)</td>
<td>283 (42)</td>
<td>19</td>
<td>8</td>
</tr>
<tr>
<td>D 349MB</td>
<td>465 (156)</td>
<td>189 (137)</td>
<td>25</td>
<td>6</td>
</tr>
<tr>
<td>E 551MB</td>
<td>880 (121)</td>
<td>884 (353)</td>
<td>47</td>
<td>17</td>
</tr>
<tr>
<td>F 31MB</td>
<td>N/A (29)</td>
<td>N/A (152)</td>
<td>52</td>
<td>13</td>
</tr>
<tr>
<td>G 798MB</td>
<td>195 (145)</td>
<td>432 (311)</td>
<td>88</td>
<td>5</td>
</tr>
</tbody>
</table>

Figure: With seeders

Figure: Without seeders
### Tribler

**Problem:**

- Most users have different upload/download speeds
- Tit-for-tat may restrict the download speed
- Solution: let your friends help you for free

---

**Bibliography**


Tribler

256 Kbps

upload

download

peer

friend

bartering

equal

for free

bartering

Alberto Montresor (UniTN)
Reading Material

  
  http://www.disi.unitn.it/~montreso/ds/papers/P2PSurvey.pdf