

Improving data access in P2P systems

Presenter: Xiaoni Lai

Roadmap

- Introduction
 - Peer-to-Peer System, Gnutella
 - Gridella
- P-Grid
 - Search Algorithm
 - Construction Algorithm
- Mapping Filename to Binary Keys
 - Trie Construction
 - Find Key on Trie
 - Uniform Distribution
- Performance Comparison between Gnutella v.s. Gridella
- Conclusion

Introduction: Peer-to-Peer System (P2P)

- Limitation of client-server-based systems
 - Network bandwidth bottleneck
- P2P System as an alternative
 - Every node/peer acts as both client and server
 - More complex searching, node organization, etc.

Introduction: Gnutella

- A P2P Success Story
- A decentralized file-sharing system
- No indexing mechanism supported
 - Search requests broadcasted over the network
 - Each recipient node scans its local database for possible answers
 - Very costly!

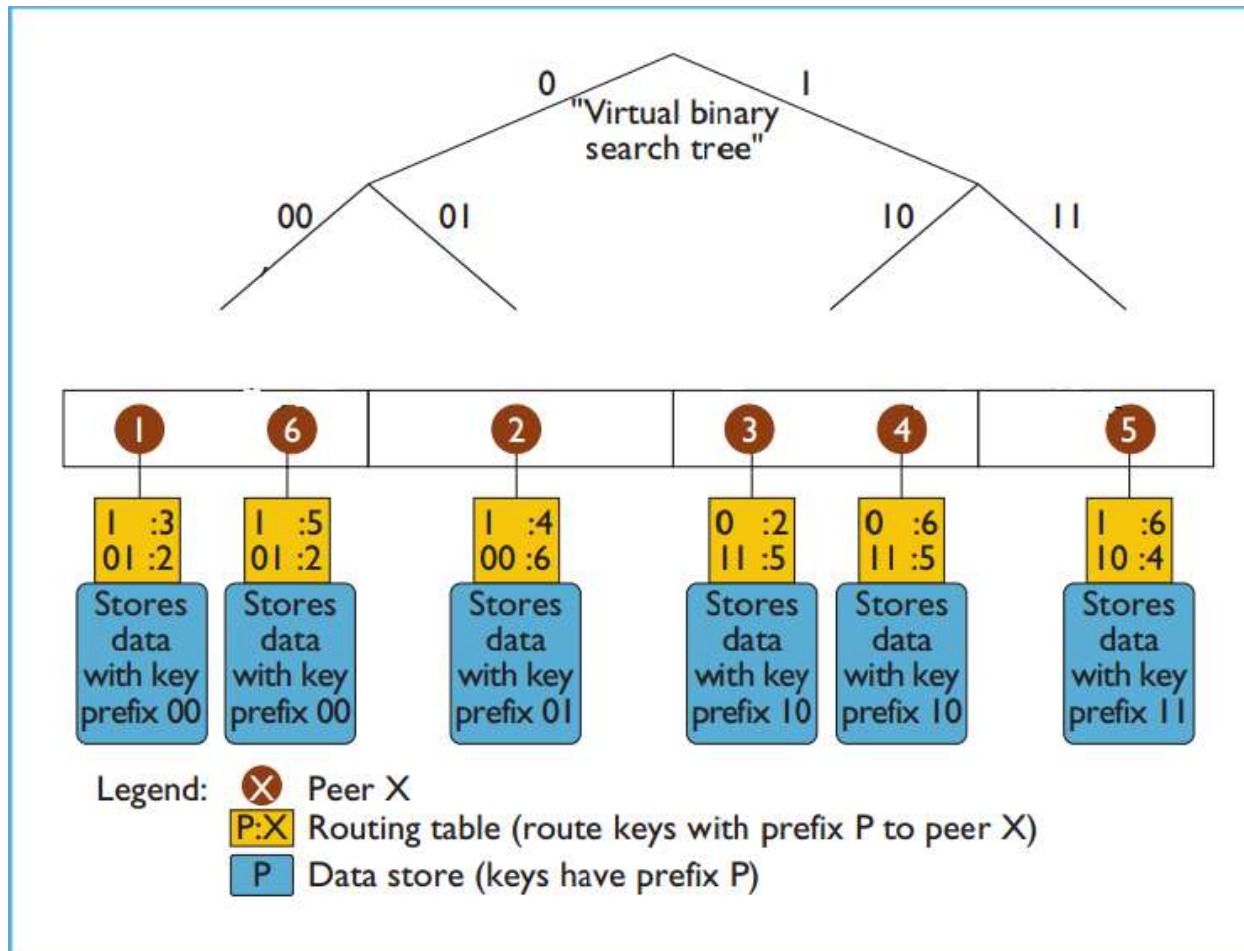
Introduction: Gridella

- Based on the Peer-Grid (P-Grid) approach
- Gnutella-compatible P2P system with a decentralized, scalable **data access** structure

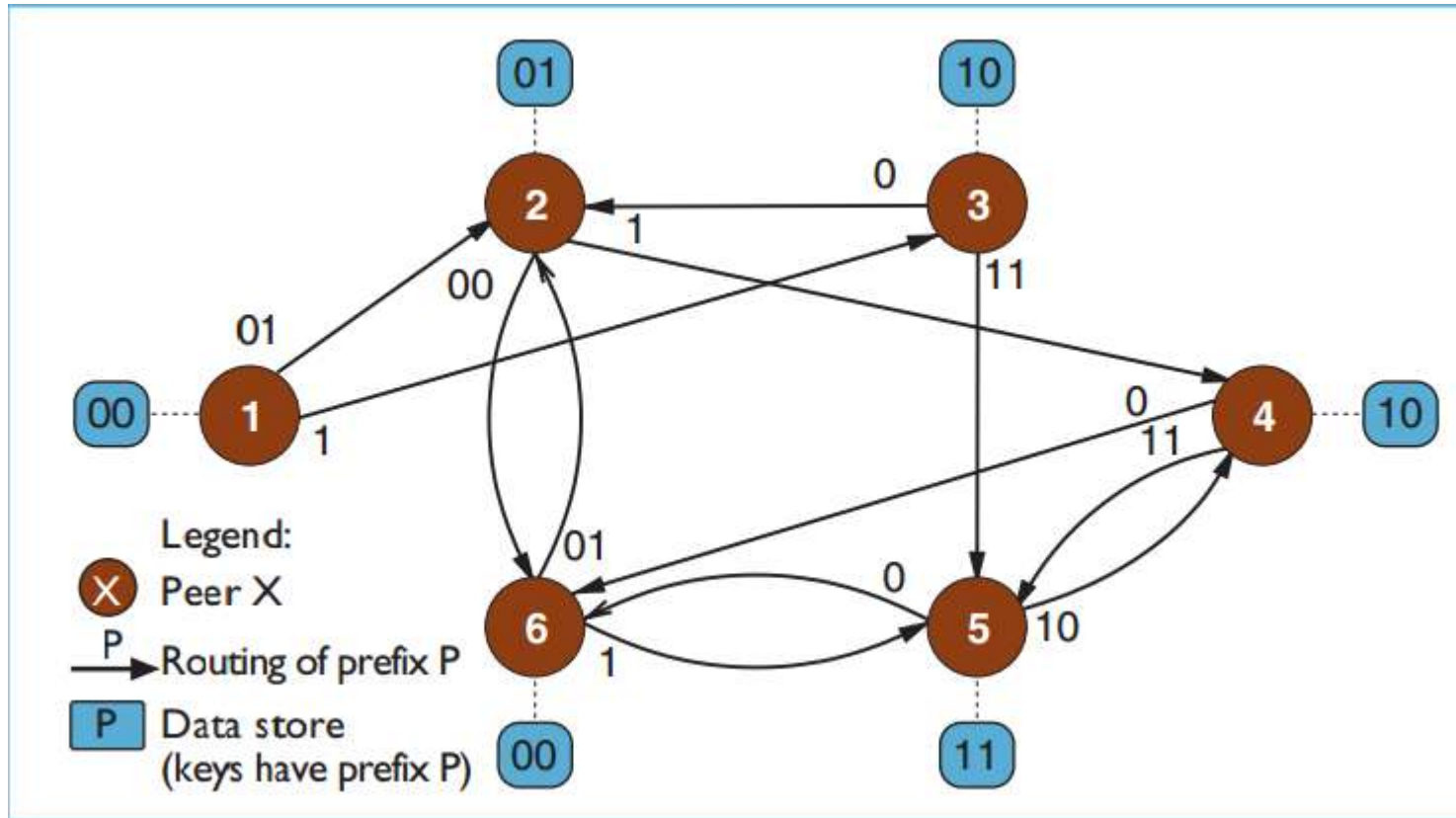
P-Grid

- A virtual binary search tree
 - Supports efficient search
- P-Grid's search structure
 - Completely decentralized
 - All peers can be entry points to the network
 - All interactions are strictly local
 - Randomized Algorithm
 - Probabilistic estimates of search request success can be given
 - Scalable and robust

P-Grid

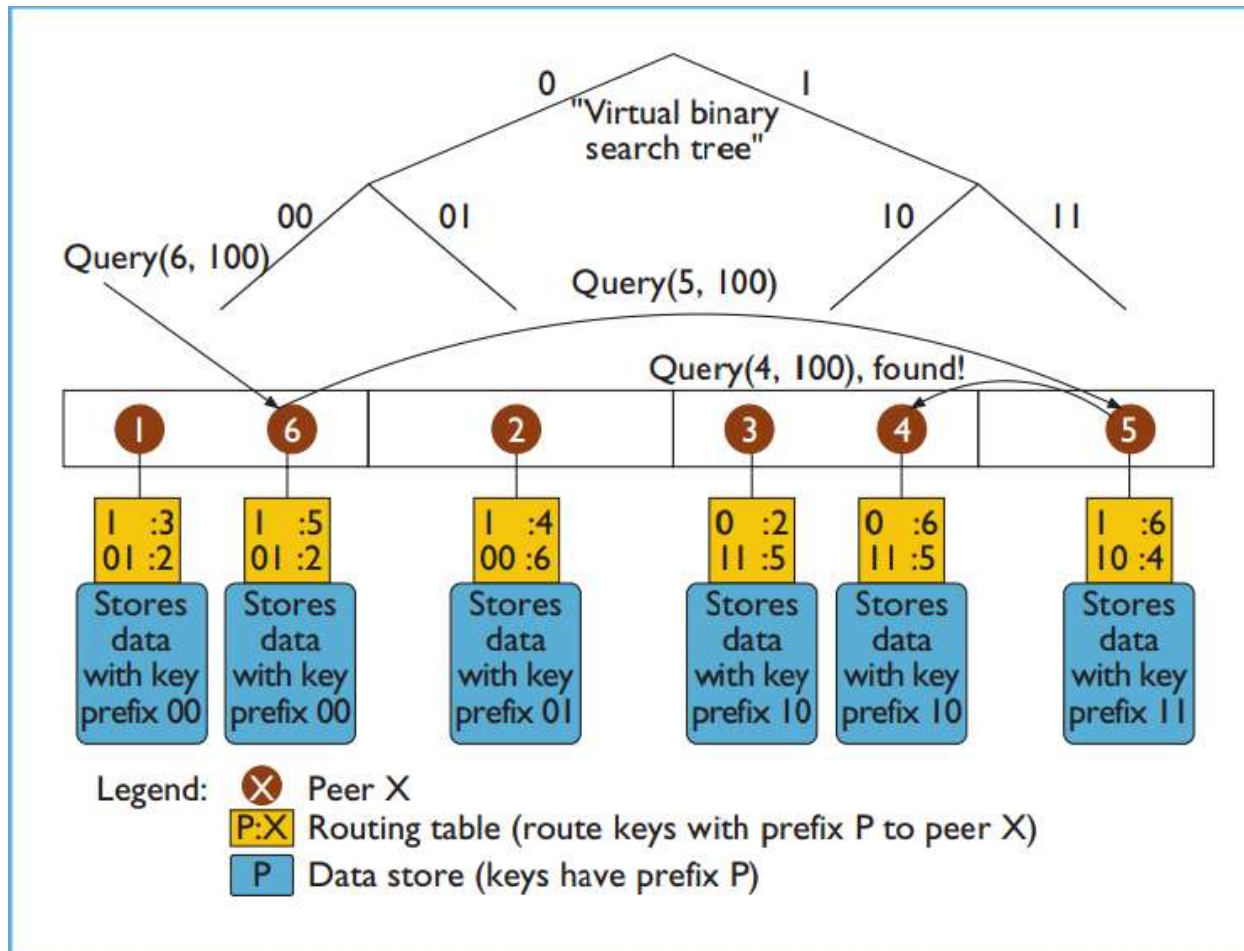


P-Grid



At least one path from any peer receiving a request to one of the peers holding the replica.

A Search Example



Search Algorithm

Search(peer with path 11, 100, 0)

```
search (peer, query, index) {
    found = NULL;          /* found: the address of the responsible peer */
    11 rempath = sub_path(path(peer), index+1, length(path(peer)));
    1  compath = common_prefix_of(query, rempath);
    IF length(compath)=length(query) OR length(compath)=length(rempath) THEN
        found = peer;
    ELSE
        IF length2(path(peer)) > index + length(compath)0+1 THEN
            00 new_query = sub_path(query, length(compath) + 1, length(query));
            refs = get_refs(index + length(compath) + 1); Get_ref(0+1+1)
            WHILE |refs| > 0 AND NOT found
                ref = random_select(refs);
                IF online(ref) Found peer with path 10
                    found = search(ref, new_query, index + length(compath));
                    Search(peer with path 10, 00, 1)
            RETURN found;
}
```

The algorithm has an input condition that the first *index* bits are truncated from the *query* string. → Optimization

Search Algorithm

Search(peer with path 10, 00, 1)

```
search (peer, query, index) {
    found = NULL; /* found: the address of the responsible peer */
    0 rempath = sub_path(path(peer), index+1, length(path(peer)));
    0 compath = common_prefix_of(query, rempath);
    IF length(compath)=length(query) OR length(compath)=length(rempath) THEN
        found = peer; 
    ELSE
        IF length(path(peer)) > index + length(compath) THEN
            new_query = sub_path(query, length(compath) + 1, length(query));
            refs = get_refs(index + length(compath) + 1);
            WHILE |refs| > 0 AND NOT found
                ref = random_select(refs);
                IF online(ref)
                    found = search(ref, new_query, index + length(compath));
    RETURN found;
}
```

The algorithm has an input condition that the first *index* bits are truncated from the *query* string. → Optimization

P-Grid Construction Algorithm

- By randomly meeting among each other, the peers
 - Successfully partition the search space
 - Retain the other peer's references for efficiently answering future search requests
 - And therefore, refine the access structure

P-Grid Construction Algorithm

- Initially, all peers are responsible for the entire search space
 - When two meet, they split the search space into two parts and each takes one half
 - Store reference of the other peer
- Similar action if both peers are responsible for the same path

P-Grid Construction Algorithm

- As soon as P-Grid develops, two scenarios occur.
- If peers whose paths share a common prefix meet
 - Initiate new exchanges by forwarding each other to their referenced peers
- If peers whose paths are in a prefix relationship meet
 - Peers with shorter path would specialize (in the opposite direction) by extending its path



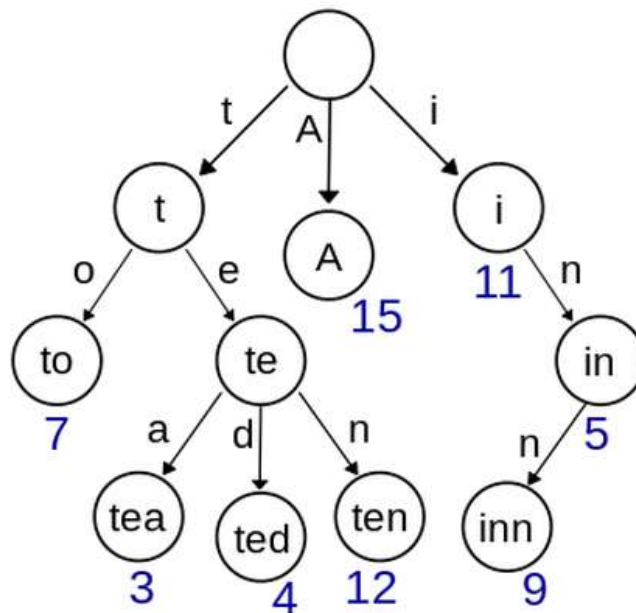
P-Grid Construction Algorithm

```
exchange(a1, a2, r) {
    determine the common prefix of path(a1) and path(a2) and its length lc;

    exchange references at the level where the paths match;
    li = length of remaining path of ai;
    /* Case 1: both paths empty, introduce new level */
    CASE l1 = 0 AND l2 = 0 AND lc < maximum possible path length
        extend path(a1) with 0 and path(a2) with 1;
        add mutual references for future search;
    /* Case 2: one remaining path empty, split shorter path */
    CASE l1 = 0 AND l2 > 0 AND lc < maximum possible path length
        extend path(a1) by one bit different to the corresp. bit in path(a2);
        update references of a1 with a2;
    /* Case 3: analogous to case 2 with roles exchanged */
    ...
    /* Case 4: recursively exchange with referenced peers */
    CASE l1 > l2 > 0 AND r < maximum recursion depth
        take a reference from a2 at the level of the common prefix;
        a1 performs a new exchange with the referenced peer
        (which shares with a1 a longer common prefix);
    /* Case 5: analogous to case 2 with roles exchanged */
    ...
}
```

Mapping Filenames to Binary Keys

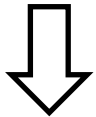
- The mapping scheme must satisfy:
 - $s_1 \text{ prefix } s_2 \Rightarrow \text{key}(s_1) \text{ prefix key}(s_2)$
- Construct a trie from a sample string database



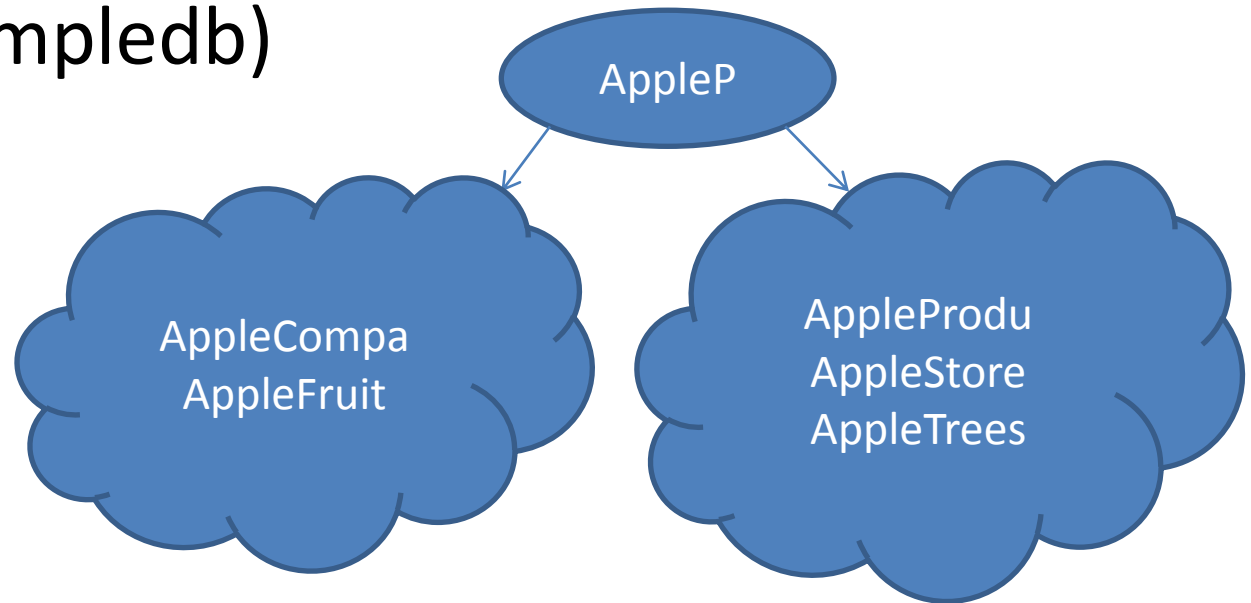
Mapping Filenames to Binary Keys

- MakeTrie(sampledb)

AppleFruit
AppleTrees
AppleCompa
AppleStore
AppleProdu



AppleCompa
AppleFruit
AppleProdu
AppleStore
AppleTrees



Length of Common Prefix: Length("Apple")

Median: "AppleProdu"

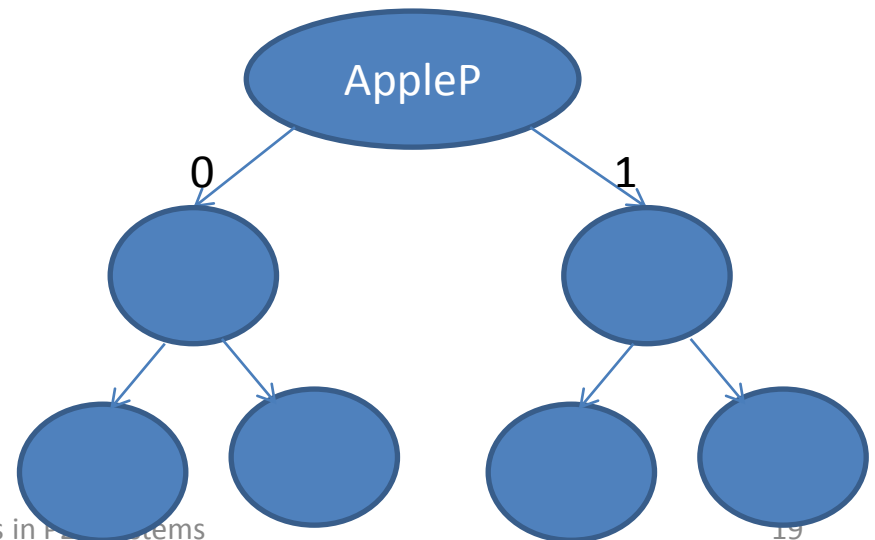
Root = Prefix of Median with Length("Apple")+1 = "AppleP"

Mapping Filenames to Binary Keys

```
MakeTrie(sampled) {  
    /* sort the sampled in lexicographical order */  
    SortLex(sampled);  
    /* find the length of the common prefix for all the strings in sampled */  
    commonprefix = Length(CommonPrefix(sampled));  
    /* We choose the median string from sampled and take the prefix of  
       length commonprefix+1 to split the sampled into two  
       approximately equal parts. It is possible to explore different  
       alternatives at this point, in order to achieve a more balanced split */  
    if Size(sampled) > MaxLeafStore then  
        mid = Prefix(sampled[Quotient(Size(sampled), 2)],  
                     Length(commonprefix + 1));  
    for j = 1 to Size(sampled) do  
        if sampled[j] is lexicographically smaller than mid then  
            lowpart = Append(lowpart, sampled[j]);  
        if sampled[j] is lexicographically greater than mid then  
            highpart = Append(highpart, sampled[j]);  
    if Size(lowpart) > MaxLeafStore then  
        left = MakeTrie(lowpart)  
    else  
        left = null;  
    if Size(highpart) > MaxLeafStore then  
        right = MakeTrie(highpart)  
    else  
        right = null;  
    root = mid;  
    TrieSet(root, left, right);  
}
```

Mapping Filenames to Binary Keys

```
FindKey (trie, filename) {  
    key = {};  
    if (trie = null) or (filename is prefix or equal to trie.root) then  
        return key;  
    else  
        if filename is lexicographically smaller than trie.root then  
            key = Append(key, 0);  
            key = Append(key, FindKey(trie.left, filename));  
        else  
            key = Append(key, 1);  
            key = Append(key, FindKey(trie.right, filename));  
    }  
}
```



Uniform Distribution

- A large sample database effectively approximates the global distribution of filenames
- 1,951 strings for sampled db; 30 MaxLeafStore; 99 keys
 - Average 342 search strings per key
 - Maximum: 798 strings to each key
 - Resulting distribution is of fairly good quality w.r.t. Uniformity.

Gridella v.s. Gnutella

- Gridella can be viewed as an extra layer on top of Gnutella

Table 1. Performance comparison of Gridella and Gnutella.

Peers	Gridella messages	Gnutella messages
20,000	61	8,744
40,000	63	26,240
60,000	65	26,240
80,000	65	78,728
100,000	68	78,728
120,000	69	78,728
140,000	68	78,728
160,000	69	78,728
180,000	69	78,728
200,000	72	78,728

Conclusion

- Simple yet successful, popular P2P systems once again prove the Internet community's ability to incubate revolutionary systems
- Still need scientific foundations
- P2P systems should extend beyond the domain of mere MP3 and image exchange
 - Future: decentralized e-commerce, mobile ad hoc networks.

Questions

- How does Gridella deal with the reality that peers are online with a low probability?
- Why must the prefix property be satisfied to ensure P-Grid of real filenames to work?
- Why do you think Gridella is able to achieve a relatively uniform load distribution for peers with respect to storage, i.e. right amount of data items responsible by each peer?
- How does data updates occur in P-Grid?

Uniform Load Distribution

- Important to P2P; otherwise it would gradually degenerate into a backbone-based system.
- Factors contributing to uniformity in Gridella
 - Mapping algorithm generates good distribution for the number of strings encoded to each key
 - Separation of peer identifier and peer's path
 - Peer's path is not determined as a priori
 - Peer's path indicate responsibilities for data with certain keys
 - The self-organizing P-Grid construction process
 - The exchange function inherently tends to balance the distribution of keys → Self-stabilizing algorithm
 - makes it adapt to a given distribution of data keys stored by the peers
 - Present data keys determine the virtual trie structure
 - Controlled Replication, where a globally constant replication factor is introduced.

Updates in P-Grid

- Randomly performing depth-first searches for peers responsible for the key multiple times and propagating the update to them
- Performing breadth-first searches for peers responsible for the key once and propagating the update to them
- Creating a list of buddies for each peer, i.e. other peers that share the same key, and propagate the update to all buddies.

Is it possible that the tree becomes up to linear depth in network size?

- This sounds like the worst case for degenerated data key distributions
- But it won't happen for a randomized selection of links to other peers in the routing tables, probabilistically the search cost in terms of messages remains logarithmic, independently of the length of the paths occurring in the virtual tree.