The main challenge in P2P computing is to design and implement a robust and scalable distributed system composed of inexpensive, individually unreliable computers in unrelated administrative domains. The participants in a typical P2P system might include computers at homes, schools, and businesses, and can grow to several million concurrent participants.

P2P systems are attractive for several reasons:

- The barriers to starting and growing such systems are low, since they usually don’t require any special administrative or financial arrangements, unlike centralized facilities;
- P2P systems offer a way to aggregate and make use of the tremendous computation and storage resources on computers across the Internet; and
- The decentralized and distributed nature of P2P systems gives them the potential to be robust to faults or intentional attacks, making them ideal for long-term storage as well as for lengthy computations.

P2P computing raises many interesting research problems in distributed systems. In this article we will look at one of them, the lookup problem. How do you find any given data item in a large P2P system in a scalable manner, without any centralized servers or hierarchy? This problem is at the heart of any P2P system. It is not addressed well by most popular systems currently in use, and it provides a good example of how the challenges of designing P2P systems can be addressed.

The recent algorithms developed by several research groups for the lookup problem present a simple and general interface, a distributed hash table (DHT). Data items are inserted in a DHT and found by specifying a unique key.

Looking Up Data in P2P Systems

By Hari Balakrishnan, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica

Designing and implementing a robust distribution system composed of inexpensive computers in unrelated administrative domains.
One might believe P2P systems are mainly used for illegal music-swapping and little else, but this would be a rather hasty conclusion.

The Lookup Problem

One might believe P2P systems are mainly used for illegal music-swapping and little else, but this would be a rather hasty conclusion.

The Lookup Problem

One might believe P2P systems are mainly used for illegal music-swapping and little else, but this would be a rather hasty conclusion.
CAN, Chord, Kademlia, Pastry, Tapestry, and Viceroy are both structured and symmetric, unlike all the other systems mentioned here. This allows them to offer guarantees while simultaneously not being vulnerable to individual node failures. They all implement the DHT abstraction.

The rest of this article discusses these recent algorithms, highlighting design points and trade-offs. These algorithms incorporate techniques that scale well to large numbers of nodes, to locate keys with low latency, to handle dynamic node arrivals and departures, to ease the maintenance of per-node routing tables, and to balance the distribution of keys evenly among the participating nodes.

A Distributed Hash Table

A hash-table interface is an attractive foundation for a distributed lookup algorithm because it places few constraints on the structure of keys or the values they name. The main requirements are that data be identified using unique numeric keys, and that nodes be willing to store keys for each other. The values could be actual data items (file blocks), or could be pointers to where the data items are currently stored.

A DHT implements just one operation: lookup(key) yields the network location of the node currently responsible for the given key. A simple distributed storage application might use this interface as follows. To publish a file under a particular unique name, the publisher would convert the name to a numeric key using an ordinary hash function such as SHA-1, then call lookup(key). The publisher would then send the file to be stored at the node(s) responsible for the key. A consumer wishing to read that file would later obtain its name, convert it to a key, call lookup(key), and ask the resulting node for a copy of the file.

To implement DHTs, lookup algorithms have to address the following issues:

- Mapping keys to nodes in a load-balanced way.
- In general, all keys and nodes are identified using an m-bit number or identifier (ID). Each key is stored at one or more nodes whose IDs are “close” to the key in the ID space.
- Forwarding a lookup for a key to an appropriate node. Any node that receives a query for a key identifier must be able to forward it to a node whose ID is “closer” to the query identifier. This rule will guarantee that the query eventually arrives at the closest node.
- Distance function. The two previous issues allude to the “closeness” of keys to nodes and nodes to each other; this is a common notion whose definition depends on the scheme. In Chord, the closeness is the numeric difference between two IDs; in Pastry and Tapestry, it is the number of common prefix bits; in Kademlia, it is the bit-wise exclusive-or (XOR) of the two IDs. In all the schemes, each forwarding step reduces the closeness between the current node handling the query and the sought key.
- Building routing tables adaptively. To forward lookup messages, each node must know about some other nodes. This information is maintained in routing tables, which must adapt correctly to asynchronous and concurrent node joins and failures.

Routing in One Dimension

A key difference in the algorithms is the data structure that they use as a routing table to provide O(log N) lookups. Chord maintains a data structure that resembles a skiplist. Each node in Kademlia, Pastry, and Tapestry maintains a tree-like data structure. Viceroy maintains a butterfly data structure, which requires information about only constant other number nodes, while still providing O(log N) lookup. A recent variant of Chord uses de Bruijn graphs, which requires each node to know only about two other nodes, while also providing O(log N) lookup. We illustrate the issues in routing using Chord and Pastry’s data structure.

Chord: Skiplist-like routing

Mapping a key to a node in a load-balanced array makes:

m - 1, lookup(key).
() , lookup(key).
, lookup(key),

Routing in One Dimension

Figure 1. A structure resembling a skiplist data structure.
Tree-like routing

1, 00, 01, 10, 11, 000, 0,
2, 6
Routing in Multiple Dimensions

Routing in Multiple Dimensions

Summary and Open Questions

Routing in Multiple Dimensions

Summary and Open Questions
In summary, these P2P lookup algorithms have been implemented in a number of applications and services. They all share the DHT abstraction, which reveals a number of issues that require further investigation to resolve. They all share the DHT abstraction, which reveals a number of issues that require further investigation to resolve.

They all share the DHT abstraction, which reveals a number of issues that require further investigation to resolve. They all share the DHT abstraction, which reveals a number of issues that require further investigation to resolve. They all share the DHT abstraction, which reveals a number of issues that require further investigation to resolve.

They all share the DHT abstraction, which reveals a number of issues that require further investigation to resolve.