What is a P2P system?

- A distributed system architecture:
  - No centralized control
  - Nodes are symmetric in function
- Larger number of unreliable nodes
- Enabled by technology improvements

Diagram:
- Internet
- Nodes connected to Internet
- Diagram illustrates a network of nodes interacting with the Internet.
P2P: an exciting social development

- Internet users cooperating to share, for example, music files
  - Napster, Gnutella, Morpheus, KaZaA, etc.
- Lots of attention from the popular press
  "The ultimate form of democracy on the Internet"
  "The ultimate threat to copy-right protection on the Internet"
How to build critical services?

- Many critical services use Internet
  - Hospitals, government agencies, etc.
- These services need to be robust
  - Node and communication failures
  - Load fluctuations (e.g., flash crowds)
  - Attacks (including DDoS)
Example: robust data archiver

• Idea: archive on other user’s machines
• Why?
  • Many user machines are not backed up
  • Archiving requires significant manual effort now
  • Many machines have lots of spare disk space
• Requirements for cooperative backup:
  • Don’t lose any data
  • Make data highly available
  • Validate integrity of data
  • Store shared files once
• More challenging than sharing music!
The promise of P2P computing

• Reliability: no central point of failure
  • Many replicas
  • Geographic distribution
• High capacity through parallelism:
  • Many disks
  • Many network connections
  • Many CPUs
• Automatic configuration
• Useful in public and proprietary settings
Distributed hash table (DHT)

- DHT distributes data storage over perhaps millions of nodes

Distributed hash table

Distributed application

put(key, data) ↓ get (key) ↓ data

lookup(key) ↓ node IP address

Lookup service

node ↓ node ↓ node ↓ node

(Archiher)

(DHash)

(Chord)
DHT distributes blocks by hashing

- DHT replicates blocks for fault tolerance
- DHT balances load of storing and serving
A DHT has a good interface

- Put(key, value) and get(key) → value
  - Simple interface!
- API supports a wide range of applications
  - DHT imposes no structure/meaning on keys
- Key/value pairs are persistent and global
  - Can store keys in other DHT values
  - And thus build complex data structures
A DHT makes a good shared infrastructure

• Many applications can share one DHT service
  • Much as applications share the Internet
• Eases deployment of new applications
• Pools resources from many participants
  • Efficient due to statistical multiplexing
  • Fault-tolerant due to geographic distribution
Many applications for DHTs

- File sharing [CFS, OceanStore, PAST, Ivy, ...]
- Web cache [Squirrel, ..]
- Archival/Backup store [HiveNet, Mojo, Pastiche]
- Censor-resistant stores [Eternity, FreeNet, ..]
- DB query and indexing [PIER, ...]
- Event notification [Scribe]
- Naming systems [ChordDNS, Twine, ..]
- Communication primitives [13, ...]

Common thread: data is location-independent
DHT implementation challenges

- Data integrity
- Scalable lookup
- Handling failures
- Network-awareness for performance
- Coping with systems in flux
- Balance load (flash crowds)
- Robustness with untrusted participants
- Heterogeneity
- Anonymity
- Indexing

Goal: simple, provably-good algorithms
1. Data integrity: self-authenticating data

- Key = SHA-1(content block)
- File and file systems form Merkle hash trees
2. The lookup problem

- Get() is a lookup followed by check
- Put() is a lookup followed by a store
Centralized lookup (Napster)

SetLoc(“title”, N4)
Publisher@N4
Key=“title”
Value=file data…

Simple, but $O(N)$ state and a single point of failure
Flooded queries (Gnutella)

Robust, but worst case $O(N)$ messages per lookup
Algorithms based on routing

- Map keys to nodes in a load-balanced way
  - Hash keys and nodes into a string of digit
  - Assign key to “closest” node
- Forward a lookup for a key to a closer node
- Join: insert node in ring

Examples: CAN, Chord, Kademlia, Pastry, Tapestry, Viceroy, Koorde, ..
Chord’s routing table: fingers
Lookups take $O(\log(N))$ hops

- Lookup: route to closest predecessor
3. Handling failures: redundancy

- Each node knows IP addresses of next $r$ nodes
- Each key is replicated at next $r$ nodes
Lookups find replicas

- Opportunity to serve data from nearby node
- Use erasure codes to reduce storage and comm overhead
**Robustness Against Failures**

- 1000 DHT servers
- Average of 5 runs
- Run *before* stabilization
- All failures due to replica failing

50% of nodes disappear but only less than 1.6% of lookups fail
4. Exploiting proximity

- Nodes close on ring, but far away in Internet
- Goal: put nodes in routing table that result in few hops and low latency
- Problem: how do you know a node is nearby? How do you find nearby nodes?
Vivaldi: synthetic coordinates

- Model the network as network of springs
- Distributed machine learning algorithm
- Converges fast and is accurate
Vivaldi predicts latency well

- PlanetLab
- RON
- NYC (+)
- Australia (●)
Finding nearby nodes

- Swap neighbor sets with random neighbors
- Combine with random probes to explore
- Provably-good algorithm to find nearby neighbors based on sampling [Karger and Ruhl 02]
Reducing latency

- Latency = lookup + download
DHT implementation summary

- Chord for looking up keys
- Replication at successors for fault tolerance
- Vivaldi synthetic coordinate system for
  - Proximity routing
  - Server selection
Conclusions

• Once we have DHTs, building large-scale, distributed applications is easy
  • Single, shared infrastructure for many applications
  • Robust in the face of failures and attacks
  • Scalable to large number of servers
  • Self configuring across administrative domains
  • Easy to program

• Let’s build DHTs …. stay tuned ….

http://project-iris.net