Peer-to-peer computing research: a fad?

Frans Kaashoek
kaashoek@ls.mit.edu

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
P2P: an exciting social development

- Internet users cooperating to share, for example, music files
  - Napster, Gnutela, 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)
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
Traditional distributed computing: client/server

- Successful architecture, and will continue to be so
- Tremendous engineering necessary to make server farms scalable and robust
Application-level overlays

- One per application
- Nodes are decentralized
- NOC is centralized

P2P systems are overlay networks without central control
Distributed hash table (DHT)

- **Application** may be distributed over many nodes
- DHT distributes data storage over many nodes
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 recent DHT-based projects

- File sharing [CFS, OceanStore, PAST, IvY, ...]
- Web cache [Squirrel, ..]
- Backup store [Pastiche]
- Censor-resistant stores [Eternity, FreeNet,..]
- DB query and indexing [Hellerstein, ...]
- Event notification [Scribe]
- Naming systems [ChordDNS, Twine, ..]
- Communication primitives [I3, ...]

Common thread: data is location-independent
CFS: Cooperative file sharing

- DHT is a robust block store
- Client of DHT implements file system
  - Read-only: CFS, PAST
  - Read-write: OceanStore, Ivy
File representation: self-authenticating data

- Key = SHA(content block)
- File and file systems form Merkle hash trees
DHT distributes blocks by hashing IDs

- DHT replicates blocks for fault tolerance
- DHT caches popular blocks for load balance
Historical web archiver

- **Goal:** make and archive a daily check point of the Web
- **Estimates:**
  - Web is about 57 Tbyte, compressed
  
- **New data per day:** 580 Gbyte
  
- **Design:**
  - 12,810 nodes: 100 Gbyte disk each and 61 Kbit/s per node
Implementation using DHT

- DHT usage:
  - Crawler distributes crawling and storage load
  - Client retrieves Web pages by hash
  - DHT replicates data for fault tolerance
Backup store

• Goal: backup on other user’s machines
• Observations
  • Many user machines are not backed up
  • Backup requires significant manual effort
  • Many machines have lots of spare disk space
• Using DHT:
  • Merkle tree to validate integrity of data
  • Administrative and financial costs are less for all participants
  • Backups are robust (automatic off-site backups)
  • Blocks are stored once, if key = sha1(data)
DHT implementation challenges

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

Goal: simple, provably-good algorithms
1. The lookup problem

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

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

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, ....
Chord’s routing table: fingers

\[
\begin{array}{c}
\frac{1}{4} \\
\frac{1}{8} \\
\frac{1}{16} \\
\frac{1}{32} \\
\frac{1}{64} \\
\frac{1}{128} \\
\end{array}
\]

N80
Lookups take $O(\log(N))$ hops

- **Lookup**: route to closest predecessor
CAN: exploit d dimensions

- Each node is assigned a zone
- Nodes are identified by zone boundaries
- Join: chose random point, split its zone
Routing in 2-dimensions

- Routing is navigating a d-dimensional ID space
  - Route to closest neighbor in direction of destination
  - Routing table contains $O(d)$ neighbors

- Number of hops is $O(dN^{1/d})$
2. Balance load

- Hash function balances keys over nodes
- For popular keys, cache along the path
Why Caching Works Well

- Only $O(\log N)$ nodes have fingers pointing to N20
- This limits the single-block load on N20
3. Handling failures: redundancy

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

- Tradeoff between latency and bandwidth [Kademlia]
4. Systems in flux

- **Lookup** takes $\log(N)$ hops
  
  If system is stable
  
  But, system is never stable!

- What we desire are theorems of the type:
  
  - In the almost-ideal state, $\ldots \log(N)\ldots$
  
  - System maintains almost-ideal state as nodes join and fail
Half-life [Liben-Nowell 2002]

- Doubling time: time for N joins
- Halfing time: time for N/2 old nodes to fail
- Half life: MIN(doubling-time, halfing-time)
Applying half life

• For any node \( u \) in any P2P networks:
  If \( u \) wishes to stay connected with high probability,
  then, on average, \( u \) must be notified about
  \( W(\log N) \) new nodes per half life

• And so on...
5. Optimize routing to reduce latency

- Nodes close on ring, but far away in Internet
- Goal: put nodes in routing table that result in few hops andow latency
“close” metric impacts choice of nearby nodes

- Chord’s numerical close and table restrict choice
- Prefix-based allows for choice
- Kademlia’s offers choice in nodes and places nodes in absolute order: close (a, b) = XOR(a, b)
Neighbor set

- From k nodes, insert nearest node with appropriate prefix in routing table
- Assumption: triangle inequality holds
Finding k near neighbors

- Ping random nodes
- Swap neighbor sets with neighbors
  - Combine with random pings to explore
- Provably-good algorithm to find nearby neighbors based on sampling
  [Karger and Ruhl 02]
6. Malicious participants

- Attacker denies service
  - Flood DHT with data
- Attacker returns incorrect data [detectable]
  - Self-authenticating data
- Attacker denies data exists [liveness]
  - Bad node is responsible, but says no
  - Bad node supplies incorrect routing info
  - Bad nodes make a bad ring, and good node joins it

Basic approach: use redundancy
Sybil attack [Douceur 02]

- Attacker creates multiple identities
- Attacker controls enough nodes to foil the redundancy

Ø Need a way to control creation of node IDs
One solution: secure node IDs

- Every node has a public key
- Certificate authority signs public key of good nodes
- Every node signs and verifies messages
- Quotas per publisher
Another solution: exploit practical byzantine protocols

- A core set of servers is pre-configured with keys and perform admission control
- The servers achieve consensus with a practical byzantine recovery protocol [Castro and Liskov '99 and '00]
- The servers serialize updates [OceanStore] or assign secure node IDs [Configuration service]
A more decentralized solution: weak secure node IDs

- ID = SHA-1 (IP-address node)
  - Assumption: attacker controls limited IP addresses

- Before using a node, challenge it to verify its ID
Using weak secure node IDS

- Detect malicious nodes
  - Define verifiable system properties
    - Each node has a successor
    - Data is stored at its successor
  - Allow querier to observe lookup progress
    - Each hop should bring the query closer
  - Cross check routing tables with random queries
- Recovery: assume limited number of bad nodes
- Quota per node ID
Philosophical questions

• How decentralized should systems be?
  • Gnutela versus content distribution network
  • Have a bit of both? (e.g., OceanStore)

• Why does the distributed systems community have more problems with decentralized systems than the networking community?
  • “A distributed system is a system in which a computer you don’t know about renders your own computer unusable”
  • Internet (BGP, NetNews)
What are we doing at MIT?

- **Building a System based on Chord**
  - Applications: CFS, Herodotus, Melody, Backup store, IvY, ...

- **Collaborate with other institutions**
  - P2P workshop, PlanetLab
  - Big ITR

- **Building a large-scale testbed**
  - RON, PlanetLab
Summary

- 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
7. Programming abstraction

- Blocks versus files
- Database queries (join, etc.)
- Mutable data (writers)
- Atomicity of DHT operations