CIS 455/555: Internet and Web Systems

Decentralized Computing

November 17, 2021
Plan for today

- Decentralized Computing
  - Unstructured Overlays
  - Structured Overlays
A New Challenge

- Suppose we want to do asynchronous communication, not waiting for responses, across a cluster! Requesting computation or data or ...?

- A challenge: distributing load across machines!
  - Master / worker configuration
  - If we just use a hierarchy, we tend to get a bottleneck at the root node

- Alternative: decentralized systems – no master at all!
Spectrum of approaches

- Centralized: Client/server
- Partly centralized: BitTorrent, Skype, ...
- Decentralized: Pastry, Chord, Gnutella
Characteristics of partly centralized systems

- Contains some infrastructure components
  - Example: Central controller that maintains a list of participating nodes

- But: Infrastructure component is not involved in resource-intensive operations
  - Example: Data is downloaded or uploaded directly to peers

- Sometimes called a "peer-assisted" system
An example

Suppose we want to ship a DVD image to 10,000 clients. How do we do this?

- **Option #1**: Server does all the work
  - Example: 1 Gbps upstream → Need about 190 hours

- **Option #2**: Let the clients help
  - 1 Mbps upstream x 10,000 = 10 Gbps!
  - Even if the server has only 1 Mbps, can finish in 19 hours!
Swarming

Client now has entire file, turns into a 'seeder'

Node that originally has the file

Fixed-size pieces
How do clients find peers to connect to?

- Clients connect to a special tracker node
- Tracker responds with the IP+port of a few other peers who are downloading the same file
  - Modern BitTorrent clients are trackerless and use a DHT instead (more about this later)

How do clients find the tracker?

- Clients begin by downloading a 'torrent file' (e.g., from a web server), which has the URL of the tracker
- Torrent file also contains a SHA1 hash of each file block
  - Why is this needed?
BitTorrent

Simplified BitTorrent session:

1. Download the 'torrent file'
2. Connect to the tracker and get a list of peers
3. Connect to the peers - initially as a 'leecher'
4. While file is not yet fully downloaded:
   - Advertise to peers which blocks are available locally
   - Request blocks from peers
   - Compare hash of downloaded blocks to hash in torrent file (why?)
5. Turn into a 'seeder', i.e., continue uploading to peers without downloading
**Incentives**

Many users would rather not upload content

- Some users pay per byte (e.g., cellular networks)
- Uploading may take bandwidth from other applications
- Upload traffic may introduce jitter or queueing delay (VoIP!)

**Danger: “Tragedy of the commons”**

- Everyone wants to download, but nobody uploads

**Idea: Provide an incentive for uploading**

- Many possible incentives (name a few!)
- BitTorrent's approach is based on reciprocity
Tit for tat

Idea: Allow requests from peers with best rate of serving the network

- Result: Everyone has an incentive to upload
- Instance of an old, successful idea
  - Goes back to Axelrod's tournament (iterated prisoner's dilemma)
- Attempts to achieve Pareto efficiency

How this is used in BitTorrent:

- At any given time, peer uploads blocks to a fixed # of other peers
- Peers are chosen based on current download (sharing) rate
- All other peers are 'choked' (no uploads) except one (to bootstrap)
- May also periodically "unchoke" random peers

Upload = share with
Download = get from
Beyond Trackers

- How might we build fully functional peer-to-peer systems?
  - Unstructured systems
  - Structured overlay networks
Plan for today

- Decentralized Computing
  - Unstructured Overlays
  - Structured Overlays
Fully decentralized systems

No dedicated nodes that are critical for the operation of the system

- No inherent bottlenecks
- Can scale very very far
- Potentially resilient to failure, attack, legal challenges

Often a high degree of symmetry

- Most functions can be performed by any node
- Example: Storing a certain file, forwarding packets, ...
- Does this mean that all nodes have to be exactly the same?
The bootstrap problem

There’s no one to track who’s in the system... so:
How does a new node join the system?
  - Partly centralized system: Can ask the 'controller' for peers
  - But in a decentralized system, there is no 'controller'!

Need to find at least one existing peer. How?
  - Idea #1: Hardcode a few IPs in the software
  - Idea #2: Keep a web page with a few current IPs
  - Idea #3: Use chat
  - Idea #4: Use DNS
  - Idea #5: Use anycast
  - ...
Overlays

Peers must somehow connect to each other

- Result: **Logical network topology**

Relationship to the physical network topology?

- Direct link in logical topology does not imply direct link in **physical topology**, or vice versa
- But link in logical topology requires path in physical topology
- Logical topology is an **overlay**
Unstructured overlay

- What should the logical topology be like?
- One option: No constraints at all
  - When a peer joins, it randomly connects to some other peers
    - How many should it connect to?
  - Result: Unstructured overlay
    - Example: Gnutella; we saw earlier
- Where should we store content? How can we locate it?
  - Name at least two ways to do each
Store+retrieve: Unstructured overlay

- Content can be stored anywhere
  - Example: On the node that inserted it
  - Maybe add a few replicas or pointers on other nodes

- To retrieve: **Flooding or random walks**
  - What should be the scope of the flood? Length of the walk?

- Pros and cons?
  - Very good at finding hay, not so good at finding needles
  - Finding unpopular objects can be expensive
  - Object may not be found even if it exists
Coordination: Epidemic protocol

- How can we disseminate information in an unstructured overlay?

- Idea: Use **epidemic protocol** ('Gossiping')
  - Source node tells a few of its peers, who in turn tell a few of their own peers, and so on
  - Similar to spread of infections or rumors in a population
  - Pro: Very simple and extremely robust
  - Con: Can be slow; difficult to target a subset of the nodes
Peers join and leave over time ('churn')
- Temporary departure: Node switched off, loses connectivity...
- Permanent departure: Reinstallation, node decommissioned...

Need overlay maintenance
- Replace links to departed nodes with new links
- Otherwise, overlay will disconnect eventually!

May need replication + replica maintenance
- If objects need to remain available despite churn
- Permanent departures destroy replicas; need to replace them
- How many replicas should we make?
Recap: Unstructured overlays

- No restrictions on which peers can connect
  - Result: Random topology

- A few techniques for unstructured overlays
  - Flooding or random walks to locate content
  - Epidemic protocol for coordination

- Both advantages and disadvantages
  - Advantages: Simple, robust, low-maintenance
  - Disadvantage: Not very efficient (flooding!)
Plan for today

- Decentralized Computing ✔
  - Unstructured Overlays ✔
  - Structured Overlays
Structured overlay

- Idea: Add structure!
  - Give each node a unique identifier
  - Use the identifier to restrict who can connect to whom
  - Example: Identifiers are from a circular ID space, and nodes form a ring (Pastry)

- Result: Can assign specific roles to nodes
  - For example, node 42 can be responsible for object 39
  - Everyone knows where to look for object 39!
The Core Idea: Hashing

- Start with a hash function with a uniform distribution of values:
  - \( h(\text{name}) \rightarrow \text{a value (e.g., 32-bit integer)} \)

- Map from values to hash buckets
  - Generally using \( \text{mod (\# buckets)} \)

- Put items into the buckets
  - May have “collisions” and need to chain

\[ h(x) \text{ values} \quad \text{buckets} \]

overflow chain
Dividing Hash Tables Across Machines

- **Simple distribution** – allocate some number of hash buckets to various machines
  - Can give this information to every client, or provide a central directory
  - Can evenly or unevenly distribute buckets
  - Lookup is very straightforward

- **A possible issue** – data skew: some ranges of values occur frequently
  - Can use dynamic hashing techniques
  - Can use better hash function, e.g., SHA-1 (160-bit key)
Issues with Conventional Hashing

- What if the set of servers holding the inverted index is dynamic?
  - Our number of buckets changes
  - How much work is required to reorganize the hash table?

- Solution: consistent hashing
Consistent Hashing: the Basis of “Structured P2P”

Intuition: a distributed hash table where the number of buckets stays constant, even if the number of machines changes

- Requires a mapping from hash entries to nodes
- Don’t need to re-hash everything if node joins/leaves
- Only the mapping (and allocation of buckets) needs to change when the number of nodes changes

Many examples: CAN, Pastry, Chord

- Chord is simplest to understand, so we’ll look at it
Basic Ideas

- We’re going to use a giant hash key space
  - SHA-1 hash: 20B, or 160 bits
  - We’ll arrange it into a “circular ring” (it wraps around at $2^{160}$ to become 0)

- We’ll actually map both objects’ keys (in our case, keywords) and nodes’ IP addresses into the same hash key space
  - “abacus” → SHA-1 → k10
  - 130.140.59.2 → SHA-1 → N12
Chord Hashes a Key to its *Successor*

- Nodes and blocks have randomly distributed IDs
  - *Successor*: node with next highest ID
Basic Lookup: Linear Time

- Lookups find the ID’s predecessor
- Correct if successors are correct

“Where is k70?”

“N80”

“Finger Table” Allows $O(\log N)$ Lookups

- Goal: shortcut across the ring – binary search
  - Reasonable lookup latency
Node Joins

- How does the node know where to go? (Suppose it knows 1 peer)
- What would need to happen to maintain connectivity?
- What data needs to be shipped around?
A Graceful Exit: Node Leaves

- What would need to happen to maintain connectivity?

- What data needs to be shipped around?
What about Node Failure?

- Suppose a node just dies?
- What techniques have we seen that might help?
Successor Lists Ensure Connectivity

- Each node stores $r$ successors, $r = 2 \log N$
- Lookup can skip over dead nodes to find objects
Objects are Replicated as Well

- When a “dead” peer is detected, repair the successor lists of those that pointed to it.
- Can take the same scheme and replicate objects on each peer in the successor list.
  - Do we need to change lookup protocol to find objects if a peer dies?
  - Would there be a good reason to change lookup protocol in the presence of replication?
Thinking of a P2P System as a Messaging System (Pastry, etc.)

- Can we set up a distributed *messaging* infrastructure using consistent hashing?

- Send message to the *owner* of a key, it returns a result!
Protecting data in P2P Systems

- Adversary's goal: Return 'bad' content
  - Studies show that unprotected systems tend to be rife with mislabeled or corrupt content (remember Gnutella?)

- Your goal: Verify that content is genuine
  - Is the result of a GET operation really the expected object?

- Idea: Make objects self-certifying
  - When inserting object O, choose key as hash(O)
  - When retrieving object, check whether key=hash(response)
  - What do we do if the object is mutable?
Controlling membership

- Who can join the decentralized system?
  - Many systems allow anyone to join

- Problem: Sybil attack
  - Adversary joins system with a large number of (physical or virtual) peers
  - Can censor objects, snoop on data, disrupt routing, ...
  - Successfully used, e.g., on the Vanish system

- How to defend a system against this?
  - Proof of work, controlled membership, ...
Topology attacks

- Suppose adversary wants to censor objects in a DHT
  - Is a defense against Sybil enough?

- Problem: Adversary can hide other nodes
  - Example: Choose keys to control an object, or to control a specific peer's routing table
    - Possible countermeasure: Certified identities
  - Example: Adversary tells the victim only about other nodes controlled by the adversary (Eclipse attack)
Recap: Attacks on KBR

- Like all systems, decentralized systems need to be protected against attacks

- We've seen three examples of such attacks:
  - Data corruption, Sybil attack, Eclipse attack
  - (There are others)

- We've also briefly discussed some defenses:
  - Self-certifying objects → Can detect data corruption
  - Proof of work, controlled membership → Sybil attack is harder
Distributed Hash Tables, Summarized

- Provide a way of deterministically finding an entity in a distributed system, without a directory, and without worrying about failure.

- Can also be a way of dividing up work: instead of sending data to a node, might send a task.
  - Note that it’s up to the individual nodes to do things like store data on disk (if necessary; e.g., using B+ Trees).
Applications of Distributed Hash Tables

- To build distributed file systems (CFS, PAST, ...)
- To distribute “latent semantic indexing” (U. Rochester)
- As the basis of distributed data integration (U. Penn, U. Toronto, EPFL) and databases (UC Berkeley)
- To archive library content (Stanford)

The “consistent hashing” scheme is used in Amazon DynamoDB for replication and data access (though it’s not P2P)