Introduction to P2P systems

CompSci 230 - UC, Irvine
Prof. Nalini Venkatasubramanian

Acknowledgements: Slides modified from
Sukumar Ghosh, U at IOWA
Mark Jelasity, Tutorial at SASO’07
Keith Ross, Tutorial at INFOCOM
P2P Systems

Use the vast resources of machines at the edge of the Internet to build a network that allows resource sharing without any central authority.

More than a system for sharing pirated music/movies.
Characteristics of P2P Systems

- Exploit edge resources.
  - Storage, content, CPU, Human presence.
- Significant autonomy from any centralized authority.
  - Each node can act as a Client as well as a Server.
- Resources at edge have intermittent connectivity, constantly being added & removed.
  - Infrastructure is untrusted and the components are unreliable.
A P2P network is an overlay network. Each link between peers consists of one or more IP links.
Overlays: All in the application layer

- **Tremendous design flexibility**
  - Topology, maintenance
  - Message types
  - Protocol
  - Messaging over TCP or UDP

- **Underlying physical network is transparent to developer**
  - But some overlays exploit proximity
Overlay Graph

Virtual edge
- TCP connection
- or simply a pointer to an IP address

Overlay maintenance
- Periodically ping to make sure neighbor is still alive
- Or verify aliveness while messaging
- If neighbor goes down, may want to establish new edge
- New incoming node needs to bootstrap

Could be a challenge under high rate of churn
- Churn: dynamic topology and intermittent access due to node arrival and failure
Overlay Graph

- **Unstructured overlays**
  - e.g., new node randomly chooses existing nodes as neighbors

- **Structured overlays**
  - e.g., edges arranged in restrictive structure
P2P Applications

- P2P File Sharing
  - Napster, Gnutella, Kazaa, eDonkey, BitTorrent
  - Chord, CAN, Pastry/Tapestry, Kademlia
- P2P Communications
  - MSN, Skype, Social Networking Apps
- P2P Distributed Computing
  - Seti@home
P2P File Sharing

Alice runs P2P client application on her notebook computer intermittently connects to Internet

Intermittently connects to Internet

Gets new IP address for each connection

Asks for “Hey Jude”

Application displays other peers that have copy of Hey Jude.

Alice chooses one of the peers, Bob.

File is copied from Bob’s PC to Alice’s notebook

While Alice downloads, other users upload from Alice.
P2P Communication

- **Instant Messaging**
- **Skype is a VoIP P2P system**

Alice runs IM client application on her notebook computer **intermittently connects to Internet**

gets new IP address for each connection

Register herself with "system"

Learns from "system" that Bob in her buddy list is active

Alice initiates direct TCP connection with Bob, then chats

P2P
P2P/Grid Distributed Processing

- seti@home
  - Search for ET intelligence
  - Central site collects radio telescope data
  - Data is divided into work chunks of 300 Kbytes
  - User obtains client, which runs in background
  - Peer sets up TCP connection to central computer, downloads chunk
  - Peer does FFT on chunk, uploads results, gets new chunk

- Not P2P communication, but exploit Peer computing power
Promising properties of P2P

- Massive scalability
- Autonomy: non single point of failure
- Resilience to Denial of Service
- Load distribution
- Resistance to censorship
Key Issues

- Management
  - How to maintain the P2P system under high rate of churn efficiently
  - Application reliability is difficult to guarantee

- Lookup
  - How to find out the appropriate content/resource that a user wants

- Throughput
  - Content distribution/dissemination applications
  - How to copy content fast, efficiently, reliably
Management Issue

- A P2P network must be **self-organizing**.
  - Join and leave operations must be self-managed.
  - The infrastructure is untrusted and the components are unreliable.
- The number of faulty nodes grows linearly with system size.
- **Tolerance to failures and churn**
  - Content replication, multiple paths
  - Leverage knowledge of executing application
- Load balancing
- Dealing with **freeriders**
  - Freerider: rational or selfish users who consume more than their fair share of a public resource, or shoulder less than a fair share of the costs of its production.
Lookup Issue

- How do you locate data/files/objects in a large P2P system built around a dynamic set of nodes in a scalable manner without any centralized server or hierarchy?
- Efficient routing even if the structure of the network is unpredictable.
  - Unstructured P2P: Napster, Gnutella, Kazaa
  - Structured P2P: Chord, CAN, Pastry/Tapestry, Kademlia
Napster

**Centralized Lookup**
- Centralized directory services
- Steps
  - Connect to Napster server.
  - Upload list of files to server.
  - Give server keywords to search the full list with.
  - Select “best” of correct answers. (ping)

**Performance Bottleneck**
- Lookup is centralized, but files are copied in P2P manner
Gnutella

**Fully decentralized lookup** for files

- The main representative of “unstructured P2P”
- Flooding based lookup
- Obviously **inefficient** lookup in terms of scalability and bandwidth
Gnutella: Scenario

**Step 0: Join the network**

**Step 1: Determining who is on the network**
- "Ping" packet is used to announce your presence on the network.
- Other peers respond with a "Pong" packet.
- Also forwards your Ping to other connected peers
- A Pong packet also contains:
  - an IP address
  - port number
  - amount of data that peer is sharing
  - Pong packets come back via same route

**Step 2: Searching**
- Gnutella "Query" ask other peers (usually 7) if they have the file you desire
- A Query packet might ask, "Do you have any content that matches the string ‘Hey Jude’?"
- Peers check to see if they have matches & respond (if they have any matches) & send packet to connected peers if not (usually 7)
- Continues for TTL (how many hops a packet can go before it dies, typically 10)

**Step 3: Downloading**
- Peers respond with a “QueryHit” (contains contact info)
- File transfers use direct connection using HTTP protocol’s GET method
Gnutella: Reachable Users (analytical estimate)

*T*: TTL, *N*: Neighbors for Query

<table>
<thead>
<tr>
<th>T=1</th>
<th>T=2</th>
<th>T=3</th>
<th>T=4</th>
<th>T=5</th>
<th>T=6</th>
<th>T=7</th>
</tr>
</thead>
<tbody>
<tr>
<td>N=2</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>10</td>
<td>12</td>
</tr>
<tr>
<td>N=3</td>
<td>3</td>
<td>9</td>
<td>21</td>
<td>45</td>
<td>93</td>
<td>189</td>
</tr>
<tr>
<td>N=4</td>
<td>4</td>
<td>16</td>
<td>52</td>
<td>160</td>
<td>484</td>
<td>1,456</td>
</tr>
<tr>
<td>N=5</td>
<td>5</td>
<td>25</td>
<td>105</td>
<td>425</td>
<td>1,705</td>
<td>6,825</td>
</tr>
<tr>
<td>N=6</td>
<td>6</td>
<td>36</td>
<td>186</td>
<td>936</td>
<td>4,686</td>
<td>23,436</td>
</tr>
<tr>
<td>N=7</td>
<td>7</td>
<td>49</td>
<td>301</td>
<td>1,813</td>
<td>10,885</td>
<td>65,317</td>
</tr>
<tr>
<td>N=8</td>
<td>8</td>
<td>64</td>
<td>456</td>
<td>3,200</td>
<td>22,408</td>
<td>156,864</td>
</tr>
</tbody>
</table>
Gnutella : Search Issue

- **Flooding based search is extremely wasteful with bandwidth**
  - A large (linear) part of the network is covered irrespective of hits found
  - Enormous number of redundant messages
  - All users do this in parallel: local load grows linearly with size

- **What search protocols can we come up with in an unstructured network**
  - Controlling topology to allow for better search
    - Random walk, Degree-biased Random Walk
  - Controlling placement of objects
    - Replication
Gnutella: Random Walk

Basic strategy
- In **scale-free graph**: high degree nodes are easy to find by (biased) random walk
  - **Scale-free graph** is a graph whose degree distribution follows a **power law**
- And high degree nodes can store the index about a large portion of the network

- **Random walk**
  - avoiding the visit of last visited node

- **Degree-biased random walk**
  - Select highest degree node, that has not been visited
  - This first climbs to highest degree node, then climbs down on the degree sequence
  - Provably optimal coverage
Gnutella: Replication

Spread copies of objects to peers: **more popular objects can be found easier**

Replication strategies
- When $q_i$ is the proportion of query for object $i$
- Owner replication
  - Results in proportional replication to $q_i$
- Path replication
  - Results in square root replication to $q_i$
- Random replication
  - Same as path replication to $q_i$, only using the given number of random nodes, not the path

But there is still the difficulty with rare objects.
KaZaA

Hierarchical approach between Gnutella and Napster

- Two-layered architecture.
- Powerful nodes (supernodes) act as local index servers, and client queries are propagated to other supernodes.
- Each supernode manages around 100-150 children
- Each supernode connects to 30-50 other supernodes

- More efficient lookup than Gnutella and more scalable than Napster
Nodes that have more connection bandwidth and are more available are designated as supernodes.

Each supernode acts as a mini-Napster hub, tracking the content (files) and IP addresses of its descendants:

- For each file: File name, File size, Content Hash, File descriptors (used for keyword matches during query)

- **Content Hash:**
  - When peer A selects file at peer B, peer A sends ContentHash in HTTP request
  - If download for a specific file fails (partially completes), ContentHash is used to search for new copy of file.
KaZaA: Parallel Downloading and Recovery

- If file is found in multiple nodes, user can select **parallel downloading**
  - Identical copies identified by **ContentHash**
- **HTTP byte-range header** used to request different portions of the file from different nodes
- **Automatic recovery** when server peer stops sending file
  - **ContentHash**
Unstructured vs Structured

**Unstructured P2P networks** allow resources to be placed at any node. The network topology is arbitrary, and the growth is spontaneous.

- **Structured P2P networks** simplify resource location and load balancing by defining a topology and defining rules for resource placement.
  - Guarantee **efficient search for rare objects**

What are **the rules???

**Distributed Hash Table (DHT)**
Hash Tables

- Store arbitrary keys and satellite data (value)
  - put(key, value)
  - value = get(key)
- Lookup must be fast
  - Calculate hash function \( h() \) on key that returns a storage cell
  - Chained hash table: Store key (and optional value) there
Distributed Hash Table

- Hash table functionality in a P2P network: lookup of data indexed by keys
- Key-hash → node mapping
  - Assign a unique live node to a key
  - Find this node in the overlay network quickly and cheaply
- Maintenance, optimization
  - Load balancing: maybe even change the key-hash → node mapping on the fly
  - Replicate entries on more nodes to increase robustness
Distributed Hash Table

Abstract “allocated array” called ID space, indexed by hash values

1 = h(k₁) = h(k₂)
2
3 = h(k₃)
4 = h(k₄)
5 = h(k₅)
6
7

Actual nodes in the network (dynamic)

2
k₁ v₁ k₂ v₂

4
k₃ v₃ k₄ v₄

7
k₅ v₅

Stored entries

consistent hashing of keys to nodes typically two step, as shown above
Structured P2P Systems

- Chord
  - Consistent hashing based ring structure
- Pastry
  - Uses ID space concept similar to Chord
  - Exploits concept of a nested group
- CAN
  - Nodes/objects are mapped into a d-dimensional Cartesian space
- Kademlia
  - Similar structure to Pastry, but the method to check the closeness is XOR function
Chord

- Consistent hashing based on an ordered ring overlay
- Both keys and nodes are hashed to 160 bit IDs (SHA-1)
- Then keys are assigned to nodes using consistent hashing
  - Successor in ID space
Chord: hashing properties

- Consistent hashing
  - Randomized
    - All nodes receive roughly equal share of load
  - Local
    - Adding or removing a node involves an $O(1/N)$ fraction of the keys getting new locations

- Actual lookup
  - Chord needs to know only $O(\log N)$ nodes in addition to successor and predecessor to achieve $O(\log N)$ message complexity for lookup
Chord: Primitive Lookup

- Lookup query is forwarded to successor.
  - one way
- Forward the query around the circle
- In the worst case, O(N) forwarding is required
  - In two ways, O(N/2)
Chord: Scalable Lookup

*i*th entry of a finger table points the successor of the key \( \text{nodeID} + 2^i \)

A finger table has \( O(\log N) \) entries and the scalable lookup is bounded to \( O(\log N) \)
Chord: Node join

- A new node has to
  - Fill its own successor, predecessor and fingers
  - Notify other nodes for which it can be a successor, predecessor of finger
- Simpler way: Find its successor, then stabilize
  - Immediately join the ring (lookup works), then modify the structure
Chord: Stabilization

- If the ring is correct, then routing is correct, fingers are needed for the speed only
- Stabilization
  - Each node periodically runs the stabilization routine
  - Each node refreshes all fingers by periodically calling `find_successor(n+2^i-1)` for a random `i`
  - Periodic cost is $O(\log N)$ per node due to finger refresh
Failed nodes are handled by
- Replication: instead of one successor, we keep $r$ successors
  - More robust to node failure (we can find our new successor if the old one failed)
- Alternate paths while routing
  - If a finger does not respond, take the previous finger, or the replicas, if close enough

- At the DHT level, we can replicate keys on the $r$ successor nodes
  - The stored data becomes equally more robust
Pastry

- Applies a sorted ring in ID space like Chord
  - Nodes and objects are assigned a 128-bit identifier
- NodeID is interpreted as sequences of digit with base $2^b$
  - In practice, the identifier is viewed in base 16.
  - Nested groups
- Applies Finger-like shortcuts to speed up lookup
- The node that is responsible for a key is numerically closest (not the successor)
  - Bidirectional and using numerical distance
Pastry: Nested group

Simple example: nodes & keys have n-digit base-3 ids, eg, 02112100101022
- There are 3 nested groups for each group
- Each node knows IP address of one delegate node in some of the other groups
- Suppose node in group 222... wants to lookup key k = 02112100210.
  - Forward query to a node in 0..., then to a node in 02..., then to a node in 021..., then so on.
Pastry: Routing table and LeafSet

### Routing table
- Provides delegate nodes in nested groups
- **Self-delegate** for the nested group where the node is belong to
- $O(\log N)$ rows $\Rightarrow O(\log N)$ lookup

### Leaf set
- Set of nodes which is numerically closest to the node
  - $L/2$ smaller & $L/2$ higher
- Replication boundary
- Stop condition for lookup
- **Support reliability and consistency**
  - Cf) Successors in Chord

**Base-4 routing table**

<table>
<thead>
<tr>
<th>Nodeld 10233102</th>
</tr>
</thead>
<tbody>
<tr>
<td>Leaf set</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>10233033</td>
</tr>
<tr>
<td>10233001</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Routing table</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>10233102</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>10-0-31203</td>
</tr>
<tr>
<td>10-2-0230</td>
</tr>
<tr>
<td>1023-0-322</td>
</tr>
<tr>
<td>10233-0-01</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>
Pastry: Join and Failure

- **Join**
  - Use routing to find numerically closest node already in network
  - Ask state from all nodes on the route and initialize own state

- **Error correction**
  - Failed leaf node: contact a leaf node on the side of the failed node and add appropriate new neighbor
  - Failed table entry: contact a live entry with same prefix as failed entry until new live entry found, if none found, keep trying with longer prefix table entries
Hash value is viewed as a point in a D-dimensional Cartesian space
- Hash value points \( \langle n_1, n_2, \ldots, n_D \rangle \).
- Each node responsible for a D-dimensional “cube” in the space
- Nodes are neighbors if their cubes “touch” at more than just a point

- Example: \( D=2 \)
- 1’s neighbors: 2, 3, 4, 6
- 6’s neighbors: 1, 2, 4, 5
- Squares “wrap around”, e.g., 7 and 8 are neighbors
- Expected # neighbors: \( O(D) \)
To get to \(<n_1, n_2, ..., n_D>\) from \(<m_1, m_2, ..., m_D>\)
- choose a neighbor with smallest Cartesian distance from \(<m_1, m_2, ..., m_D>\) (e.g., measured from neighbor’s center)

- e.g., region 1 needs to send to node covering X
- Checks all neighbors, node 2 is closest
- Forwards message to node 2
- Cartesian distance monotonically decreases with each transmission
- Expected # overlay hops: \(\frac{(D N^{1/D})}{4}\)
To join the CAN:
- find some node in the CAN (via bootstrap process)
- choose a point in the space uniformly at random
- using CAN, inform the node that currently covers the space that node splits its space in half
  - 1st split along 1st dimension
  - if last split along dimension i < D, next split along i+1st dimension
  - e.g., for 2-d case, split on x-axis, then y-axis
- keeps half the space and gives other half to joining node

The likelihood of a rectangle being selected is proportional to its size, i.e., big rectangles chosen more frequently
CAN Failure recovery

- View partitioning as a binary tree
  - Leaves represent regions covered by overlay nodes
  - Intermediate nodes represents “split” regions that could be “reformed”
  - Siblings are regions that can be merged together (forming the region that is covered by their parent)
CAN Failure Recovery

- Failure recovery when leaf S is removed
  - Find a leaf node T that is either
    - S’s sibling
    - Descendant of S’s sibling where T’s sibling is also a leaf node
  - T takes over S’s region (move to S’s position on the tree)
  - T’s sibling takes over T’s previous region
Kademlia : BitTorrent DHT

- For each node, files, keywords, deploy SHA-1 hash into a 160 bits space.
- Every node maintains information about files, keywords “close to itself”.
- The closeness between two objects measure as their bitwise XOR interpreted as an integer.
- \( D(a, b) = a \text{ XOR } b \)
Kademlia: Binary Tree

Space of 160-bit numbers

Subtrees for node 0011...

Each subtree has k buckets (k delegate nodes)
Kademlia: Lookup

When node 0011...... wants search 1110......

O(log N)
P2P Content Dissemination
Content dissemination

Content dissemination is about allowing clients to actually get a file or other data after it has been located.

Important parameters:
- Throughput
- Latency
- Reliability
Problem Formulation

- **Least time to disseminate:**
  - Fixed data $D$ from one seeder to $N$ nodes

- **Insights / Axioms**
  - Involving end-nodes speeds up the process (Peer-to-Peer)
  - Chunking the data also speeds up the process

- **Raises many questions**
  - How do nodes find other nodes for exchange of chunks?
  - Which chunks should be transferred?
  - Is there an optimal way to do this?
Optimal Solution in Homogeneous Network

- Least time to disseminate:
  - All M chunks to N-1 peers
- Constraining the problem
  - Homogeneous network
  - All Links have same throughput & delay
  - Underlying network fully connected (Internet)

- Optimal Solution (DIM): $\log_2 N + 2(M-1)$
  - Ramp-Up: Until each node has at least 1 chunk
  - Sustained-Throughput: Until all nodes have all chunks

- There is also an optimal chunk size

References:
- Ganesan, P. On Cooperative Content Distribution and the Price of Barter. ICDCS 2005
Example Working of Optimal Solution
Practical Content dissemination systems

- Centralized
  - Server farms behind single domain name, load balancing

- Dedicated CDN
  - CDN is independent system for typically many providers, that clients only download from (use it as a service), typically http
  - Akamai, FastReplica

- End-to-End (P2P)
  - Special client is needed and clients self-organize to form the system themselves
  - BitTorrent(Mesh-swarm), SplitStream(forest), Bullet(tree+mesh), CREW(mesh)
Akamai

- Provider (eg CNN, BBC, etc) allows Akamai to handle a subset of its domains (authoritative DNS)
- HTTP requests for these domains are redirected to nearby proxies using DNS
  - Akamai DNS servers use extensive monitoring info to specify best proxy: adaptive to actual load, outages, etc
- Currently 20,000+ servers worldwide, claimed 10-20% of overall Internet traffic is Akamai
- Wide area of services based on this architecture
  - availability, load balancing, web based applications, etc
Decentralized Dissemination

Tree:
- Intuitive way to implement a decentralized solution
- Logic is built into the structure of the overlay

However:
- Sophisticated mechanisms for heterogeneous networks (SplitStream)
- Fault-tolerance Issues

Mesh-Based (Bittorrent, Bullet):
- Multiple overlay links
- High-BW peers: more connections
- Neighbors exchange chunks

Robust to failures
- Find new neighbors when links are broken
- Chunks can be received via multiple paths

Simpler to implement
BitTorrent

- Currently 20-50% of internet traffic is BitTorrent
- Special client software is needed
  - BitTorrent, BitTyrant, µTorrent, LimeWire ...
- Basic idea
  - Clients that download a file at the same time help each other (i.e., also upload chunks to each other)
  - BitTorrent clients form a swarm: a random overlay network
BitTorrent: Publish/download

- **Publishing a file**
  - Put a "*.torrent" file on the web: it contains the address of the tracker, and information about the published file
  - Start a tracker, a server that
    - Gives joining downloaders random peers to download from and to
    - Collects statistics about the swarm
  - There are “trackerless” implementations by using Kademlia DHT (e.g. Azureus)

- **Download a file**
  - Install a bittorrent client and click on a "*.torrent" file
BitTorrent: Overview

File.torrent:
- URL of tracker
- File name
- File length
- Chunk length
- Checksum for each chunk (SHA1 hash)

Seeder – peer having entire file
Leecher – peer downloading file
BitTorrent: Client

- Client first asks 50 random peers from tracker
  - Also learns about what chunks (256K) they have
- Pick a chunk and tries to download its pieces (16K) from the neighbors that have them
  - Download does not work if neighbor is disconnected or denies download (choking)
  - Only a complete chunk can be uploaded to others
- Allow only 4 neighbors to download (unchoking)
  - Periodically (30s) optimistic unchoking: allows download to random peer
    - important for bootstrapping and optimization
  - Otherwise unchokes peer that allows the most download (each 10s)
BitTorrent: Tit-for-Tat

- Tit-for-tat
  - Cooperate first, then do what the opponent did in the previous game
- BitTorrent enables tit-for-tat
  - A client unchokes other peers (allow them to download) that allowed it to download from them
  - Optimistic unchocking is the initial cooperation step to bootstrapping
What chunk to select to download?

Clients select the chunk that is **rarest** among the neighbors (Local decision)

- Increases diversity in the pieces downloaded; Increase throughput
- Increases likelihood all pieces still available even if original seed leaves before any one node has downloaded entire file

Except the first chunk

- Select a random one (to make it fast: many neighbors must have it)
BitTorrent: Pros/Cons

- **Pros**
  - Proficient in utilizing partially downloaded files
  - Encourages diversity through “rarest-first”
    - Extends lifetime of swarm
  - Works well for “hot content”

- **Cons**
  - Assumes all interested peers active at same time; performance deteriorates if swarm “cools off”
  - Even worse: no trackers for obscure content
Overcome tree structure – SplitStream, Bullet

- Tree
  - Simple, Efficient, Scalable
  - But, vulnerable to failures, load-unbalanced, no bandwidth constraint

- SplitStream
  - Forest (Multiple Trees)

- Bullet
  - Tree(Metadata) + Mesh(Data)

- CREW
  - Mesh(Data, Metadata)
SplitStream

- Forest based dissemination
- Basic idea
  - Split the stream into K stripes (with MDC coding)
  - For each stripe create a multicast tree such that the forest
    - Contains interior-node-disjoint trees
    - Respects nodes’ individual bandwidth constraints
- Approach
  - On the Pastry and Scribe(pub/sub)
SplitStream : MDC coding

- Multiple Description coding
  - Fragments a single media stream into $M$ substreams ($M \geq 2$)
  - $K$ packets are enough for decoding ($K < M$)
  - Less than $K$ packets can be used to approximate content
    - Useful for multimedia (video, audio) but not for other data
    - Cf) erasure coding for large data file
SplitStream: Interior-node-disjoint tree

- Each node in a set of trees is interior node in at most one tree and leaf node in the other trees.
- Each substream is disseminated over subtrees.
SplitStream : Constructing the forest

- Each stream has its groupID
  - Each groupID starts with a different digit
- A subtree is formed by the routes from all members to the groupId
  - The nodeIds of all interior nodes share some number of starting digits with the subtree’s groupId.
- All nodes have incoming capacity requirements (number of stripes they need) and outgoing capacity limits
Layers a mesh on top of an overlay tree to increase overall bandwidth

Basic Idea

- Use a tree as a basis
- In addition, each node continuously looks for peers to download from
- In effect, the overlay is a tree combined with a random network (mesh)
Two phases

**Collect phase**: using the tree, membership info is propagated upward (random sample and subtree size)

**Distribution phase**: moving down the tree, all nodes are provided with a random sample from the entire tree, or from the non-descendant part of the tree
Bullet: Informed content delivery

- When selecting a peer, first a similarity measure is calculated
  - Based on summary-sketches
- Before exchange missing packets need to be identified
  - Bloom filter of available packets is exchanged
  - Old packets are removed from the filter
    - To keep the size of the set constant
- Periodically re-evaluate senders
  - If needed, senders are dropped and new ones are requested
Gossip-based Broadcast

Probabilistic Approach with Good Fault Tolerant Properties
- Choose a destination node, uniformly at random, and send it the message
- After $\log(N)$ rounds, all nodes will have the message w.h.p.
- Requires $N \times \log(N)$ messages in total
- Needs a ‘random sampling’ service

Usually implemented as
- Rebroadcast ‘fanout’ times
- Using UDP: Fire and Forget

BiModal Multicast (99), Lpbcast (DSN 01), Rodrigues’04 (DSN), Brahami ‘04, Verma’06 (ICDCS), Eugster’04 (Computer), Koldehoefe’04, Periera’03
Gossip-based Broadcast: Drawbacks

Problems
- More faults, higher fanout needed (not dynamically adjustable)
- Higher redundancy \(\rightarrow\) lower system throughput \(\rightarrow\) slower dissemination
- Scalable view & buffer management
- Adapting to nodes’ heterogeneity
- Adapting to congestion in underlying network
### CREW: Preliminaries

**Metadata**

<table>
<thead>
<tr>
<th>File Attributes</th>
<th>Checksum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name, MimeType, Size</td>
<td></td>
</tr>
<tr>
<td>Chunk-1</td>
<td></td>
</tr>
<tr>
<td>Chunk-2</td>
<td></td>
</tr>
<tr>
<td>Chunk-i</td>
<td></td>
</tr>
<tr>
<td>Chunk-M</td>
<td></td>
</tr>
</tbody>
</table>

---

CREW (Concurrent Random Expanding Walkers) Protocol

Basic Idea: Servers ‘serve’ data to only a few clients
- Who in turn become servers and ‘recruit’ more servers

- Split data into chunks
  - Chunks are concurrently disseminated through random-walks

- Self-scaling and self-tuning to heterogeneity
What is new about CREW

- No need to pre-decide fanout or complex protocol to adjust it
  - Deterministic termination
  - Autonomic adaptation to fault level (More faults $\rightarrow$ more pulls)
- Scalable, real-time and low-overhead view management
  - Number of neighbors as low as $\log(N)$ (expander overlay)
  - Neighbors detect and remove dead node $\rightarrow$ disappears from all nodes’ views instantly
  - List of node addresses not transmitted in each gossip message
- Use of metadata plus handshake to reduce data overhead
  - No transmission of redundant chunks

- Handshake overloading
  - For ‘random sampling’ of the overlay
  - Quick feedback about system-wide properties
  - Quick adaptation

- Use of TCP as underlying transport
  - Automatic flow and congestion control at network level
  - Less complexity in application layer
- Implemented using RPC middleware
CREW Protocol: Latency, Reliability

<table>
<thead>
<tr>
<th>RapID</th>
<th>Information Reintegration Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chunk Forwarding Module</td>
<td>Neighbor Maintenance Module</td>
</tr>
</tbody>
</table>

CORBA-based Middleware (ICE)

Network / OS

Completion Time (s) vs. Number of Nodes

Completion Time (s) vs. Loss Rate (%)
Fast Replica

- Disseminate large file to large set of edge servers or distributed CDN servers
- Minimization of the overall replication time for replicating a file $F$ across $n$ nodes $N_1, \ldots, N_n$.
- File $F$ is divided into $n$ equal subsequent files: $F_1, \ldots, F_n$, where $\text{Size}(F_i) = \frac{\text{Size}(F)}{n}$ bytes for each $i = 1, \ldots, n$.
- Two steps of dissemination
  - Distribution and Collection
FastReplica: Distribution

- Origin node $N_0$ opens $n$ concurrent connections to nodes $N_1, \ldots, N_n$ and sends to each node the following items:
  - a distribution list of nodes $R = \{N_1, \ldots, N_n\}$ to which subfile $F_i$ has to be sent on the next step;
  - subfile $F_i$. 
FastReplica : Collection

After receiving $F$, node $N$ opens $(n-1)$ concurrent network connections to remaining nodes in the group and sends subfile $F$ to them.
FastReplica: Collection (overall)

Each node $N_i$ has:

- $(n - 1)$ outgoing connections for sending subfile $F_i$,
- $(n - 1)$ incoming connections from the remaining nodes in the group for sending complementary subfiles $F_1, \ldots, F_{i-1}, F_{i+1}, \ldots, F_n$. 
FastReplica : Benefits

- Instead of typical replication of the entire file $F$ to $n$ nodes using $n$ Internet paths, FastReplica exploits $(n \times n)$ different Internet paths within the replication group, where each path is used for transferring $1/n$-th of file $F$.

- **Benefits:**
  - The impact of congestion along the involved paths is limited for a transfer of $1/n$-th of the file,
  - *FastReplica* takes advantage of the upload and download bandwidth of recipient nodes.