Collective Memory: Long-term storage and Sharing of PDFs

Ryan Kwon '17
Williams College
CS 339

1. INTRODUCTION

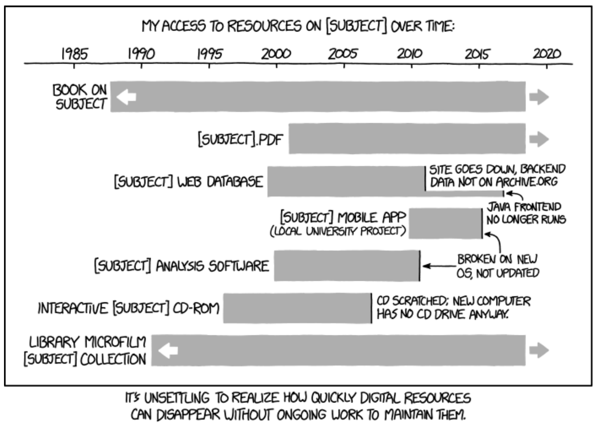
MY ACCESS TO RESOURCES ON [SUBJECT] OVER TIME:
1985
1990
1995
2000
2005
2010
2015
2020
BOOK ON SUBJECT
[SUBJECT].PDF
[SUBJECT] WEB DATABASE
SITE GOES DOWN, BACKEND DATA NOT ON ARCHIVE.ORG
[SUBJECT] MOBILE APP (LOCAL UNIVERSITY PROJECT)
JAVA FRONTEND NO LONGER RUNS
[SUBJECT] ANALYSIS SOFTWARE
BROKEN ON NEW OS, NOT UPDATED
INTERACTIVE [SUBJECT] CD-ROM
CD SCRATCHED; NEW COMPUTER HAS NO CD DRIVE ANYWAY.
LIBRARY MICROFILM [SUBJECT] COLLECTION
IT'S UNSETTLING TO REALIZE HOW QUICKLY DIGITAL RESOURCES CAN DISAPPEAR WITHOUT ONGOING WORK TO MAINTAIN THEM.

Figure 1. Scroll-over text: I spent a long time thinking about how to design a system for long-term organization and storage of subject-specific informational resources without needing ongoing work from the experts who created them, only to realized I'd just reinvented libraries.

Collective Memory was rationally justified after the fact by the above XKCD comic. Practically, our motivation is drawn primarily from the desire to build a cool decentralized system.

Collective Memory is a decentralized system that prioritizes availability and longevity of uploaded files. For us this means that, once a file is uploaded and disseminated, the system is designed to make sure that file never goes away so long as the network has sufficient capacity to hold it. From the perspective of a single node, once they discover a file they should be able to access that file in perpetuity. Collective Memory is also a system that is able to self-organize into a shallow tree, with subtrees/subnetworks corresponding to local autonomous systems. Our goal of persistence creates some additional concerns however: how do we verify that the files we're keeping forever are "good"? How do we mitigate abuse of the system and prevent overloading the system? How should we handle resource discovery?

This paper addresses the above questions among others in our discussion of the Collective Memory (CM) system. Section 1 is this introduction. Section 2 will discuss node types, their roles, and their available protocols. Section 3 will describe the overall network topology and architecture. Section 4 will discuss the election procedure when a Shepherd dies. Section 5 will discuss our redundancy strategy and how we ensure file persistence and availability. Section 6 will discuss our evaluation of the CM system. Section 7 will briefly discuss some related work, and Section 8 will discuss future work. Section 9 will conclude.

2. Node Types and Protocols

We support two node types, a normal Node and a Shepherd, which acts as a supernode.

Every node is identified with three attributes: an IP address, a port number, and a nodeId. The nodeId is a randomly generated positive integer between 0 and $2^{31}-1$, and is used to verify the Shepherd as a sender in some protocol requests (you can treat this as analogous to public/private key signing). The IP address and port refers to a node's server, which receives and processes TCP requests sent by nodes in the system.

2.1. Node

A Node is a normal node in the system. Every node is part of a Shepherd's flock or subnetwork/autonomous system. This normal node is able to GET files from the network for personal use, PROPOSE content for upload into the network, and PING their shepherd to retrieve data about the network.

For a typical node, the key data structures are:

- **files:** All network files known to this node.
- **peers:** All other network nodes known to this node, including their Shepherd.
- **myShepherd:** The shepherd for this node. Irrelevant if the node is a Shepherd.

The protocols available to a normal node are:

- **GET(file):** Retrieves a file and downloads it for the Node's personal use. The GET request is routed through the Shepherd. Specifically, the Shepherd receives the GET request and forwards it to a random node that holds the specified file. It's possible that the Shepherd forwards the request to a node that's silently died and hasn't yet been detected by the Shepherd as being dead. As such, there's no guarantee that the GET request succeeds, but the cost is simply re-trying the request.
- **PROPOSE(file):** Proposes a local file for uploading into the network. The proposed file gets sent to the Shepherd, who automatically downloads it into a specified directory. The Shepherd is then responsible for manually checking the file, and ACCEPTing it or REJECTing for upload. It's worth noting that perfectly legitimate files may be rejected by a Shepherd, and undesirable content may be accepted. This is seen as an acceptable risk. Every node is also capped to a fixed number of proposals to try to mitigate abuse. If a Shepherd detects any additional proposals from the same node, their content will be automatically rejected and won't be downloaded.
- **PING():** Pings the Shepherd. The Node informs the Shepherd of the following: their IP address, their port, their nodeId, and the list of files they're storing for the network. The Shepherd then responds with: a list of peers who are in the same subnetwork and the list of files available in the subnetwork. Pings are also used to identify when the Shepherd has died.

2.2. Shepherd

A Shepherd is a supernode that manages their local subnetwork. The Shepherd is able to do everything a normal node can do, but is also responsible for curating local content by ACCEPTing/REJECTing proposed files, and manages file redundancy through MANDATEs. Beyond these two responsibilities, the Shepherd is also very knowledgeable about their subnetwork in order to allow us to provide efficient file retrieval and sharing.

The key data structures that are relevant to the Shepherd are:

- **flock:** Which maps Node -> Node Data
- **networkFiles:** Which maps File -> List of nodes which hold that file.
- **numProposals:** Which maps Node -> Number of times that node has proposed.

The protocols available to a Shepherd are:

- **ACCEPT/REJECT(file):** Accepts or rejects a proposed file. The Shepherd is expected (but does not need to) manually inspect the specified file. On ACCEPTing, the file is considered a Network file, and is then managed for replication as all other network files. The file also then becomes available for personal downloading by any node in the subnetwork. REJECTing deletes the file, though the proposing node can propose the same file again. Shepherds keep track personally of how many times every node has proposed in order to mitigate abuse. They also do not share what files have been proposed, so a new Shepherd will have no memory of proposing nodes or any files that have been proposed. In a situation like that, the expection is that nodes will simply re-propose to the new Shepherd.
- **MANDATE(file, specified node, nodeId, node holding file):** MANDATE specifies a particular node (node A) to download a specific file for storage from another node (node B). The Shepherd includes node A's nodeId in order to verify that this request comes from their shepherd. Node A then requests the file from node B, and downloads the file into a specified directory for long-term storage.

Additionally, when the Shepherd receives a PING from a node, the Shepherd attaches a "time to live" on that node's data. Periodically, the Shepherd looks through every node in its subnetwork, and considers the nodes that have exceeded their "time to live" as dead, and no longer depends on their stored files. After these nodes have been removed, the Shepherd redistributes replicated copies of stored files to ensure that they have sufficient redundancy.

2.3. Motivations for Shepherds

There were three concerns that encouraged the Shepherd model.

- **Efficiency**: Shepherds may be used to cache or route files from other subnetworks, and their existence imposes a tree strucure on the Collective Memory network graph. Most nodes will only know information relevant to their AS/subnetwork. Since we route every get request through the Shepherd, the maximum possible distance in terms of node-hops between nodes in our network is 5. (Node A -> Shepherd A -> Shepherd B -> Node B -> Node A). We do not currently support inter-subnetwork communication though, so we'll discuss in the Future Work section. We do support load balancing by routing the GET requests through the Shepherd however.

- **Simplicity**: Coordinating efficient and effective replication among a bunch of equals sounds like a nightmare. Having a single node responsible for coordinating this is much simpler. Most nodes then only need to keep track of what information they have, and don't need to worry about other nodes. The Shepherd can aggregate relevant information, and make efficient decisions from a relatively centralized location.

- **Security**: Promising longevity on random files uploaded by total randos is also a nightmare. Having a shepherd will hopefully reduce the number of bad files that gets uploaded to the network. Furthermore, having a bunch of nodes respond to you when you say "Hello! I'd like to join!" also struck me as a great DDoS resource. Having only the shepherds respond should drastically cut down on network hogging.

3. Network Topology and Architecture

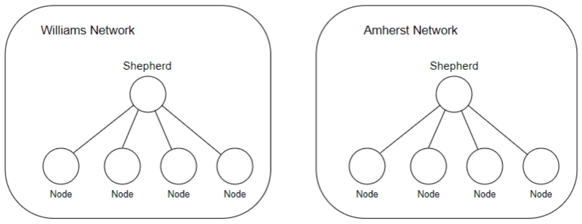

Figure 2. Williams and Amherst would have their own subnetworks. Every subnetwork is headed by a single Shepherd.

A subnetwork is a small and mostly discrete section of the larger network. Every subnetwork is headed by a single Shepherd, and they are designed to correlate strongly with local autonomous systems (AS). Figure 2 shows an example of two subnetworks. In this case, Williams and Amherst, having their own AS's, would also be their own subnetworks.

We try to enforce this local tree structure by controlling how nodes join the network.

3.1. Joining the Network

We provide two methods for joining the network:

Method one involves multicast. Specifically, every shepherd waits at a specified multicast address where they listen for join requests. These join requests consist of just the joining node's (node A) IP address and Port. Once they receive a join request, each Shepherd sends a unicast message to node A listing their IP address and port. After some period of waiting for responses, node A looks through all shepherds it's heard about, and finds its shepherd by comparing IP prefixes. A shepherd which shares node A's IP prefix is presumed to be the shepherd of node A's autonomous system/subnetwork, and node A joins that subnetwork. If no such shepherd is found, node A becomes a shepherd and waits at the multicast address.

However, multicast isn't reliable in IPv4. Packets may be dropped, and may not make it across routers. So we fall back on a second method if no responses are had at the multicast address.

The second method is the "Direct Join" method. Every node (node A) is provided with some hardcoded IP addresses/ports, or may spec-

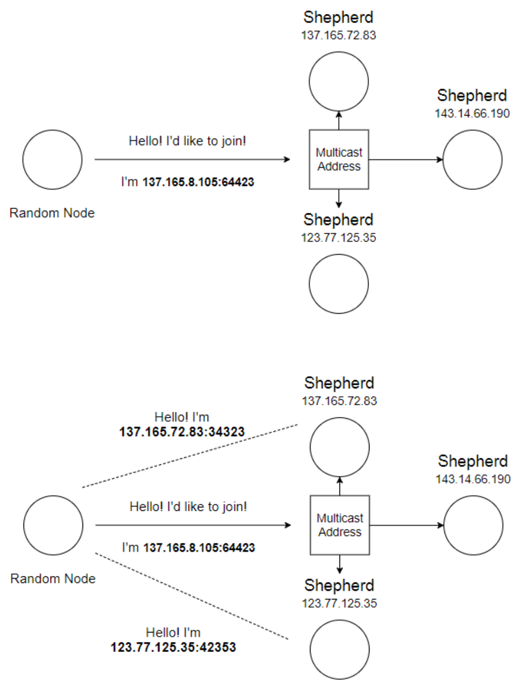

Figure 3. Illustration of a Multicast join. The top half of the diagram shows the new node sending a message to the Multicast address. The second half of the diagram shows some unicast messages from the Shepherds to the new node.

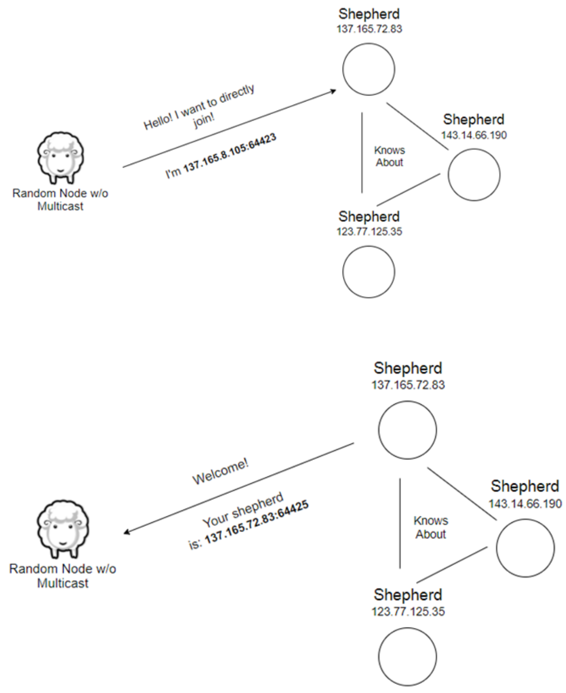

Figure 4. Illustration of a direct join. The top half shows a new node directly contacting some node. The contacted node does not need to be a Shepherd. The bottom half shows the response by the contacted node.

ify an IP address and port. If the node successfully connects to a node (node B), then node B looks through all shepherds it knows about, and informs node A of its shepherd. If node B doesn't know the shepherd of node A's subnetwork, node B responds with its own shepherd.

Note that the "Direct Join" method may break our autonomous system property. Nodes from other AS's may Direct Join into another subnetwork. We discuss a simple solution in the Future Work section.

4. Election Procedure

Shepherds serve for life, but on death we expect the subnetwork to be able to reorganize itself to recreate the tree structure and any data that was in the old network. Our goals in the election procedure are to ensure that exactly one shepherd is selected, that all legitimate nodes reconize the new shepherd, and that the selection process is at least a little random to minimize the odds that a specified malicious node becomes shepherd.

We achieve all of the above through an algorithm inspired by Paxos [**1**].

Specifically, we do this:

1. If a Shepherd does not respond to 3 consecutive pings (with a short wait time between each one), then the node (node A) assumes the Shepherd is dead.

2. Node A then looks through its list of peers and finds a peer with the lexicographically highest IP address. That peer becomes node A's Nominated Shepherd (node AS).

3. Node A then goes to each of its peers, and tells them "The shepherd is dead. My nominated shepherd is is node AS." Each peer (node B) then responds to A if and only if they've also detected that their shepherd is dead. If they have, they find their lexicographically highest known peer (for node B, call this node BS) and compare AS and BS.

4. Node B then chooses the higher peer as their nomination, and responds to node A informing them of the higher peer. If this differs from node A's highest peer, node A adjusts.

5. All nodes continue to communicate with each other in rounds, and eventually all nodes share knowledge of the highest peer in the network by IP address.

6. They then all nominate that node (node SS). Node SS then records that it's "won" an election round by receiving votes from every other peer. Node SS then sits for several election rounds. If every node continues to vote for node SS in those rounds, Node SS declares itself the shepherd and informs every other node that it's become shepherd.

7. Every node that receives a Shepherd declaration verifies it's the node they nominated. If so, they accept that node as the new Shepherd. If it isn't the node they nominated, they ignore the declaration.

Figure 5 shows an example of how two nodes share information on who the highest node in the network by IP address is.

To summarize the election procedure: the nodes communicate with each other to determine the highest node. They do this in rounds, where each node proposes to every other node the highest

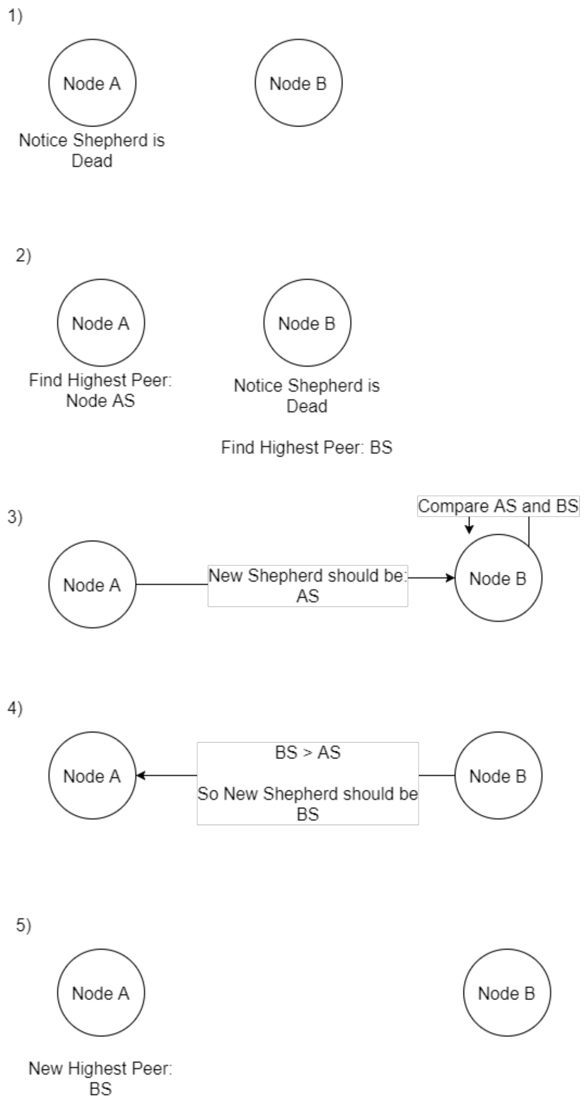

Figure 5. A brief overview of part of the Election procedure. This is the "Information Dissemination" stage, where nodes communicate with each other. Eventually, all nodes in the network will have knowledge of the highest node in the network by IP Address, and they'll all nominate that node.

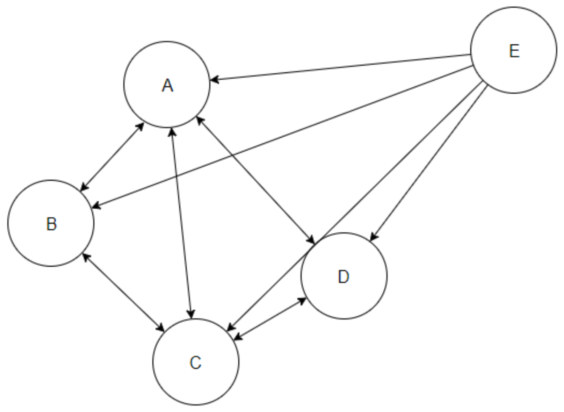

Figure 6. In this case, the network is not a complete digraph. Nodes A, B, C, and D all have knowlege of each other, but no knowledge of Node E. Node E knows about all older nodes because it must have received a ping response shortly before the Shepherd died to be part of the network at all.

node they know of and adjust this proposal based on new information. These election rounds continue until all the nodes agree on a highest node. The proposed highest node then waits to see if it unanimously wins several more elections (to allow for edge cases where some nodes may not be known by the rest of the network, and so may join the election late). If the proposed node continues to win elections, it declares itself the winner and notifies everyone else that the election is over.

4.1. Edge Cases in the Election Procedure

Note that we're not always guaranteed that the highest IP node in the network becomes elected. Specifically because the network graph may not be a complete digraph (some nodes may not have knowledge of other nodes). In this case, a new node may ping the Shepherd, get knowledge of all previous nodes, and then the Shepherd may die before sharing the new node's existence to any other nodes. We then get a graph like figure 6.

In this case, if Node E was the highest IP address, it's possible that Node E may not notice that the Shepherd is dead until Nodes A-D have already completed an election. Because they can't account for the unknown Node E, they elect the highest IP address node among them. Node E may then try to trigger an election procedure, but will be ignored by the other nodes as they believe their newly elected shepherd is alive. We deliberately have Nodes A-D ignore Node E's election procedure request in order to reduce the possibility of malicious action. Node E then doesn't have a Shepherd, and must rejoin the network to gain knowledge of the new Shepherd.

5. File Redundancy

File redundancy is dealt with strictly on the level of subnetworks. We do this primarily because we do not currently have strong safeguards against malicious nodes creating isolated subnetworks, and then rejoining the larger network. Additionally, nodes and Shepherds themselves do not have significant information or control over files in other subnetworks, but in the future may request them in order to incorporate them in their own subnetwork.

We assume that only 50% of nodes in a given subnetwork will experience simultaneous failures. The Shepherd deals with this by periodically removing dead nodes and going through all network files to ensure they're replicated on atleast 50% + 1 living nodes.

However, as each subnetwork is correlated with an autonomous system, it's reasonable to assume that a scenario where 50% of the nodes die is equally likely as a scenario where 100% of the nodes dying. We aggressively store some hard state in order to rebuild the subnetwork after such an event.

5.1. Hard State and Soft State

Whenever a node receives a file as part of a MANDATE, or a Shepherd receives a PROPOSEd file, the receiving node writes relevant file metadata to disk. This metadata includes the file name, the file path, and an object reference to the file itself. If a node goes offline and then comes back online, it's able to rebuild its **storedFiles** data structure and report that back to its Shepherd. From there, the Shepherd can include the file in its replication strategy. In other words, so long as a single node holding a specified file comes back to a network, the network is able to reconstitute itself.

6. Evaluation

Overall, the theoretical structure of the network provides several things. Latency and download times are largely irrelevant as most downloads are within the same autonomous system. Downloads from outside the autonomous system should be a rare fraction of downloads in the future, and isn't made more inefficient by costly weird routing. Load balancing is done by the Shepherd routing GET requests to a random node that holdes the file in the same system, and the entire subnetwork can be seen as a cache for all files available to a resident node. Additionally, assuming our simultaneous failure rate isn't broken, files should remain available.

We discuss here however some tests done to verify correctness of our implementation. Unfortunately, the complexity of our implementation makes it a little unwieldy to spin up new nodes automatically, but manual tests were done.

6.1. Protocol Correctness

Our implementation provides a command line interface (CLI) to access all major protocols. All of these were manually tested for both normal nodes and Shepherds, and can be verified to work. Though the MANDATE command cannot be triggered manually by a Shepherd, it has been observed to work.

6.2. Election Procedure

The election procedure as discussed in an earlier section was experimentally verified to work with the simplest case of 2 nodes, 3 nodes, and 5 nodes. In all cases, the lexicographically highest node was able to become Shepherd, as expected. In all cases, the amount of time it took for the election to conclude was around 1 minute (from the time that all nodes recognized that the shepherd had died). This is primarily driven by forced wait times. Information is disseminated completely after 1-2 rounds.

HOWEVER. The average node runs about 4 threads: the main CLI thread, the server thread, the client thread (for sending messages), and a monitoring thread. (The Shepherd runs an additional thread to manage the subnetwork). As we increase the number of nodes, race conditions may sometimes occur (concurrent modifications, attempting to reuse the client without disconnecting from an existing connection, e.t.c.), potentially causing failure in one of the threads. Fortunately (or unfortunately), these incidents are somewhat sporadic. These issures are known, and will be fixed in the future.

6.3. Redundancy

Redundancy was tested in the 1 node, 2 node, 3 node, and 5 node cases. In all cases, the shepherd correctly MANDATEd $\frac{n}{2}+1$ nodes hold any known file. Newly proposed files were also correctly replicated.

7. Related Work

The Paxos algorithm largely inspired the election procedure, though there are some differentiating factors due to our particular use case [**1**]. For example, every node participates equally in the "election," rather than having specified roles. However, our use of IP lexicographic order was directly borrowed from Paxos's use of disjoint sets of positive integers for proposal priority.

Raft, another consensus algorithm, also provided some inspiration for the election procedure [**2**]. Specifically, we also took a strongly leader-centric approach to generating consensus during normal use. Raft also has an election procedure to generate this leader, though theirs is majority-based and candidates are evaluated based on being "up-to-date" in their use-case. We eschewed the majority-based election as it seemed unnecessary, and "up-to-date"ness doesn't have a good analogue in our system.

Finally, we borrow several metrics from a case study of Gnutella [**3**]. Specifically, we also use IP prefixes to proxy the underlying DNS and physical network structure. We do this specifically to provide strong performance, but also because they seemed natural boundaries for subnetworks.

There are other decentralized systems which informed our design and creation of Collective Memory, though the above three examples are the most significant by far.

8. Future Work

The current implementation does not allow communication across subnetworks. In the future, we would like Shepherds to gossip with each other in order to propogate information about files in other subnetworks. Additionally, Shepherds should also convey some of this information to nodes in their own network. This is to allow the nodes to request files from other subnetworks, and also to ensure that they can continue communication with other subnetworks if the Shepherd dies. Although the multicast solution would make discovering other Shepherds straightforward, a unicast measure to fall back on would be needed for IPv4.

Additionally, you may also notice that our ”Direct Join” method may break our locality property, as nodes from other AS's may Direct Join into our subnetwork. A simple solution to this would be **Partitioning**: Shepherds should periodically look through nodes in their subnetwork, and split off nodes which share a foreign IP prefix if there are enough of them to create a reasonably sized subnetwork. Coupled with the above, the Shepherd should also remember the newly created subnetwork and continue communicating with them.

There are also a variety of ”robustness” measures that our current implementation lacks due to time constraints. Race conditions are possible that may result in fatal errors for the running thread, and these may force a restart of the Collective Memory node. Some subprocesses are also relatively fragile, in that they are expected to usually work, but it would boggle the mind if they worked consistently all the time in their current state.

9. Conclusion

Collective Memory is a decentralized system for uploading, sharing, and retrieving files, with the goal of persisting uploaded files for as long as possible. The system is able to organically organize itself into a shallow tree structure with a Shepherd as a ”supernode.” Every subtree (called a subnetwork or flock) contains exactly one Shepherd who manages redundancy for files, and these subnetworks correspond with autonomous systems. Because we focus solely on PDFs, and because our network topology corresponds strongly with the underlying physical network, performance appears great, and file persistence is as robust as expected.

10. Thoughts and Reflection

I did enjoy working on this project a lot, but I think two changes would have improved my experience. It would have been nice to have an additional week to work on the final project, but I admit I'm not sure what could be shortened... though the Hadoop lab may be doable within a single week? I think it would also have been nice to have more leeway in the presentation time! In my case, I think 15-20 minutes would have been very comfortable, while 10 felt rather cramped.

References

[1] Lamport, Leslie. ”Paxos made simple.” ACM Sigact News 32.4 (2001): 18-25.

[2] Ongaro, Diego, and John K. Ousterhout. ”In search of an understandable consensus algorithm.” USENIX Annual Technical Conference. 2014.

[3] Ripeanu, Matei. ”Peer-to-peer architecture case study: Gnutella network.” Peer-to-Peer Computing, 2001. Proceedings. First International Conference on. IEEE, 2001.