Collective Memory: Long-term storage and Sharing of PDFs

Ryan Kwon '17
Williams College
CS 339

1. INTRODUCTION

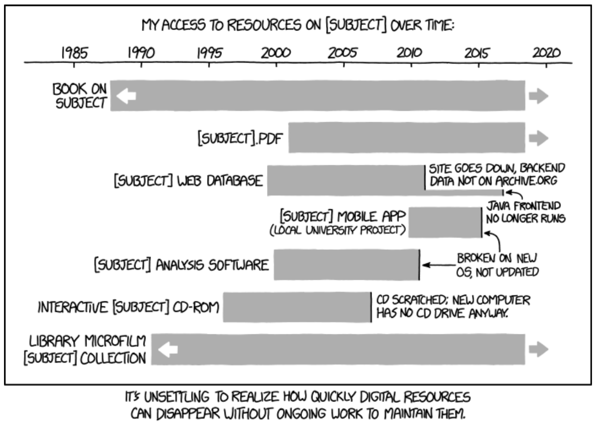
MY ACCESS TO RESOURCES ON [SUBJECT] OVER TIME:
1985 1990 1995 2000 2005 2010 2015 2020
BOOK ON SUBJECT
[SUBJECT].PDF
[SUBJECT] WEB DATABASE
SITE GOES DOWN, BACKEND DATA NOT ON ARCHIVE.ORG
[SUBJECT] MOBILE APP (LOCAL UNIVERSITY PROJECT)
JAVA FRONTEND NO LONGER RUNS
[SUBJECT] ANALYSIS SOFTWARE
BROKEN ON 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.

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. 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 files, and their existence imposes a tree strucure on the Collective Memory network graph. Most nodes will only know information relevant to their AS/subnetwork.

- **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.

- **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

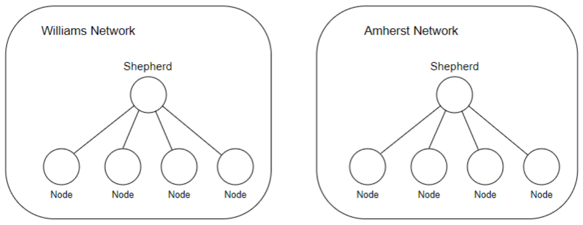

Figure 2. Williams and Amherst would have their own subnetworks. Every subnetwork is headed by a single Shepherd.

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 specify 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 this further in Future Work. 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.

4. Election Procedure:

5. Evaluation

6. Related Work

7. Future Work

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 this further in Future Work. 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.

8. Conclusions

9. Progress so far

- **Node/Network discovery Implemented:** We use multicast at a specified address and port to find network nodes. If no responses, we assume the network is dropping multicast packets (common in IPv4, though I think IPv6 supports multicast), so we fall back on a list of hard coded IP addresses and ports to ping. If none of them are reachable, we set up our own network.
- **Shepherd/Supernode model is integrated:** This is explained in a later section.

10. Things left to do

- **You should be able to specify an IP address and port to connect to.**
- **CLI interface**
- **Need to implement file-sending:** I think this should be a cinch. At that point, all the other things are pretty small.
- **Need to set up replication strategy:** This should be reasonably simple though. Most PDFs we're interested in will be fairly small (1-10 MBs likely), so we can make a reasonable guess on how many nodes will fail at once, and then replicate enough times such that we can always recover a copy if those nodes all fail.
- **Networks need to be able to reorganize themselves, so that they more logical sense:** Specifically, we aim to make subnetworks around Autonomous Systems (AS) where most of the logic will happen. This is logically simple, but I just need to write it up.
- **Need to think more on how to keep a network in the same AS from breaking up.**
- **GET and PROPOSE protocols need to be implemented**

11. Features we won't have

We won't have a "search" functionality. Although if you're ever exposed to a file, you should continue to have access to it while the relevant subnetworks are big enough to hold it. Uploaded file metadata and where they may be found are disseminated through a gossip-like protocol.

12. Node Protocol Commands

- GET: Get's a specified file given its metadata. The file is downloaded to a specific directory on the user's machine.
- PROPOSE: Propose that a file be uploaded. The proposed file goes to the shepherd, who needs to manually accept it. There is a cap on the number of files a node can propose.

13. Shepherd Model: Supernodes

The Shepherd is a supernode, gaining additional responsibilities and powers over other nodes in their AS/"flock." Every AS in a Collective Memory graph has a single shepherd (but if there are two Collective Memory networks in the same AS with no knowledge of each other, they will each have their own shepherd). For example, I'm currently the shepherd for the Williams network. Specifically, shepherds have these powers and responsibilities:

- Accept or reject "Proposed" files for upload from nodes in their AS.
- Replicating uploaded files among nodes in their AS for redundancy.
- Sit in the multicast address and accept new nodes in their AS into the network.
- Introducing nodes in their AS to each other, so that the shepherd can be replaced if she/he ever dies.
- Accept or reject files from another shepherd.

13.1. Subtleties

This section contains some extra subtleties on Shepherd behavior. This is mostly to remind myself. It will be rewritten in a more understandable format in the future.

- Isolated nodes cannot create their own nonsense by creating a network by themselves, making themselves a shepherd, and then joining the bigger network. File metadata will contain information on known shepherds when accepted. When "proposed" to another shepherd, the receiving shepherd will compare the list of shepherds extant at the time to their list of known shepherds, and reject if the file seems to have been accepted in intentionally isolated conditions.

- When a shepherd dies, nodes in its AS will become aware as their pings won't get responses. These pings are randomly distributed to occurr every 1-10 minutes. They'll then look at the multicast address to see if the new shepherd is there. If not, they become a shepherd and sit at the multicast address. If two nodes become shepherds of the same AS, they'll find each other, and the shepherd with the lexicographically lower IP address resigns and becomes a normal node. This is also to help ensure that it's difficult to become shepherd. The position is life-long, and when lost, goes to the person with the highest IP address. Something that's hopefully not super-easy to just take up.

14. Thoughts

Also Jeannie, I want to comment that I'm really, really enjoying this project. I'm having a lot of fun building this system.