Distributed Systems

Project Report
Gnutella: Consistency
Team 32

Kritin Maddireddy (2022101071)
Kandi Jayanth Reddy (2022101038)

November 27, 2024

INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY

HYDERABAD

1 Problem Statement

The Gnutella protocol, a decentralized peer-to-peer (P2P) file-sharing protocol, relies on distributed servants (peer entities functioning as both client and server). We have implemented consistency as an additional feature on top of this protocol.

2 Implementation approach

- ***Technologies:***
  - **Programming Language:** Rust for core components
  - **Version control:** Git

Our solution involves incorporating distributed consistency mechanisms into Gnutella's existing descriptor framework (Ping, Pong, Query, QueryHit, and Push).

3 Gnutella Implementation

In order to implement the Gnutella system, we have referred to v0.4 and v0.6 of the Gnutella Protocol, and ultimately, we have implemented v0.4 of this protocol.

Whichever files the user has on their system are the files that each node can access. We define a text file which contains the paths to all the files that they plan on exposing to a node. When the node first starts up, we pass this text file to it and it checks if each of the files specified in this text file actually exist, and if it has permissions to read from those files. If yes, then it stores those file paths in an array. Implicitly, the file's indices are simply their positions in the array.

Every node has (up to) three threads that are responsible to perform different functions. However, only the first two threads are maintained during the execution of the protocol.

- Thread 1 is responsible to listen for incoming connection requests (the `GNUTELLA CONNECT` message), and to create a TCP stream to whichever node that's willing to connect if it is not overburdened (it responds with `GNUTELLA OK`). Else, it provides a bunch of alternate IP addresses to the node that it can connect to. Once a connection is made, we maintain an array of all open connections to a particular node, and we have a TCP stream open to each of its connected nodes.

- Thread 2 is responsible for checking if there are any messages that have to be handled in the open TCP stream from any of its connected nodes, and is further responsible to handle those messages accordingly, sequentially.

- Thread 3 is responsible for creating a TCP stream to whichever node in the Gnutella network it connects to initially. Once that is done, this thread hands the stream over to Thread 2 for it to handle, and gracefully exits.

Initially, we define a *Descriptor Header*, which is common for all messages. This header keeps track of the kind of message being sent, and most importantly, the TTL and the number of hops that the message has already been sent through. Every single message uses this header.

Ping, *Pong*, *Query*, *QueryHit*, and *Push* are the five message types that we have implemented (in accordance with the protocol). For each of them, we implement the same packets as defined in the Gnutella v0.4 protocol. Their implementation details are as follows.

3.1 Ping

After a servent connects to a node in the network, it sends a *Ping* request to the node that it is connected to. This node is then responsible for forwarding the Ping request to other nodes in the network that it is connected to, depending on the TTL and Hops in the Descriptor Header.

3.2 Pong

To ensure **Network View Consistency**, we enforce the condition that every single node that receives a *Ping* request must necessarily respond with a *Pong* request to identify itself (unlike the Gnutella protocol). Each node will forward whatever *Pong* messages it received from the nodes that it is connected to, along with its own *Pong* message to the node that it received the *Ping* message from, with the same message ID as the *Ping* message. This way, the original servent receives multiple *Pong* messages depending on the network topography and the TTL. Essentially, the same path as *Ping* is followed while returning the *Pong* message.

3.3 Query

This follows the exact same approach in terms of path finding to *Ping*. However, the Search Criteria in the packet that we have implemented is an exact filename match.

3.4 QueryHit

This follows the exact same approach in terms of path finding to *Pong*. Whichever filename matches exactly to the search query mentioned in the *QueryHit* is returned exactly as is to the node that it received the message from. If there are multiple files, their info is stored in the Result Set according to the protocol. The initial querying node may thus receive multiple *QueryHit* messages, each with multiple entries in their Result Set.

3.5 Push

Once the querying node has all the file indices and locations, it will send a *Push* request to whichever one of them directly, out of the Gnutella network, and will keep doing so and will keep receiving the file contents from that node via HTTP until it chooses to stop. It can request to obtain any number of those files whose details it now possesses via the *QueryHit* message approach.

4 Consistency Ideas and Implementation

Firstly, in order to speed up performance of our system, and to ensure that a file gets served across as much of the network as possible, we have implemented caching of downloaded files. However, cache consistency now became a key concern. So, we have decided to implement eventual consistency of caches.

In order to enforce consistency on this protocol, we have implemented the following.

4.1 Network View Consistency

When a node receives a *Ping* message, according to the Gnutella protocol, "A servent receiving a Ping descriptor may elect to respond with a Pong descriptor ...", however, we strictly require the servent to respond with a *Pong* message. This way, network view consistency is achieved, in the sense that according to the TTL and Hops in the *Descriptor Header*, every node that receives a *Ping* request will definitely be discovered by the initially requesting servent.

4.2 File Consistency

When a node retrieves the file that it has requested from another node, it stores the file locally, essentially creating a copy of the original, and stores it in it's cache memory for as long as this node is alive. It also keeps track of where it obtained the original file from. Now, when another node requests for the same file via a *Query* request, if this node receives the *Query*, it can choose to directly respond with a *QueryHit* message. If the requesting node sends a *Push* request to this node with the cached file, the node first checks whether the cache is up to date or not by checking the last modified time of the file with the node that sent it the file originally. If it is up to date, then this node directly returns the file. If not, then it first fetches the updated file from the original sender, and then updates this file and finally returns it to the requesting node. This way, eventual consistency of cache is ensured.

If multiple files have caches, then updates will be propagated across them when a user requests a file from one cache that has data from another cache.

5 Assumptions

- If the user wants to download multiple files from a *QueryHit*, they have to perform the same *Query* multiple times.

6 Analysis

We have compared the non-caching version of Push to the eventually consistent caching version of Push, as a function of TTL for both the number of messages and the success rate.

We have done this over three different network structures, on 8 nodes and averaged results over 15 files.

In order to achieve this, we have used Bash, Python and our own Rust program to log these values.

6.1 Network Structure 1 (Linear)

6.1.1 Network Diagram

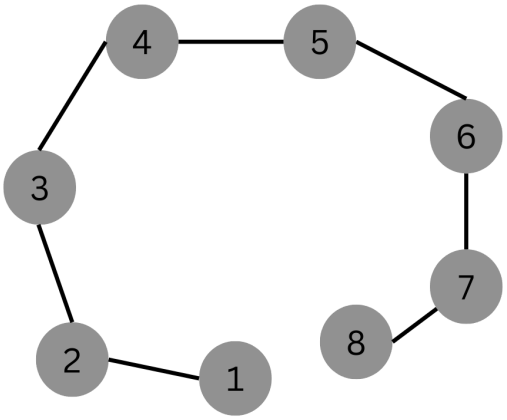
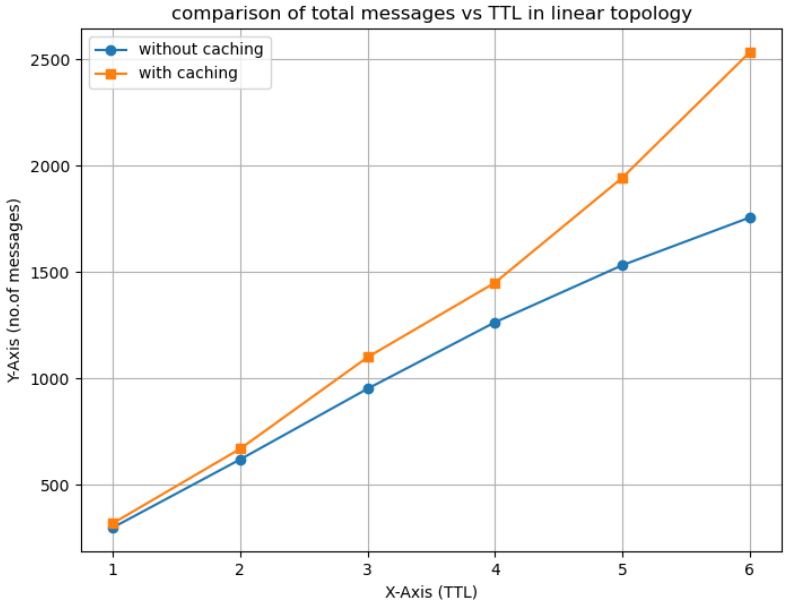
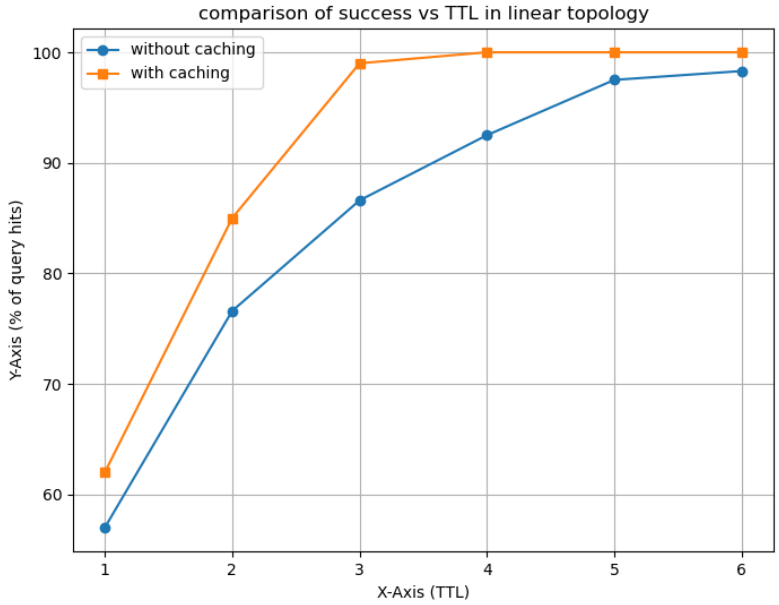


Figure 1: Linear Topology

6.1.2 Number of messages as a function of the number of message hops (TTL)



6.1.3 The success rate as a function of the number of message hops (TTL)



6.2 Network Structure 2 (Star)

6.2.1 Network Diagram

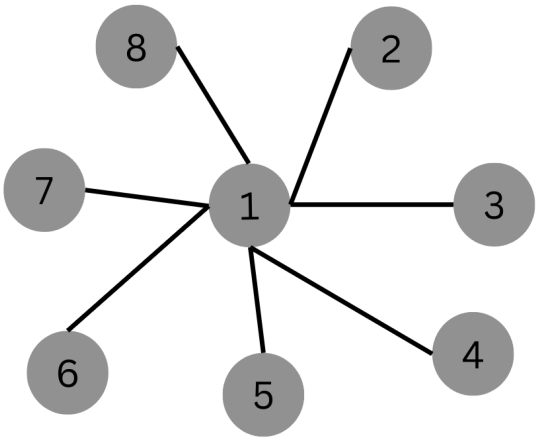
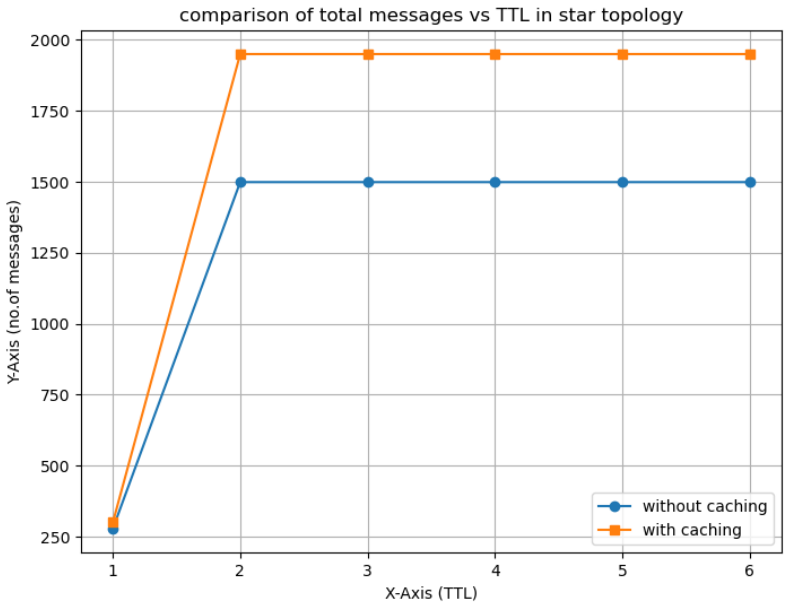
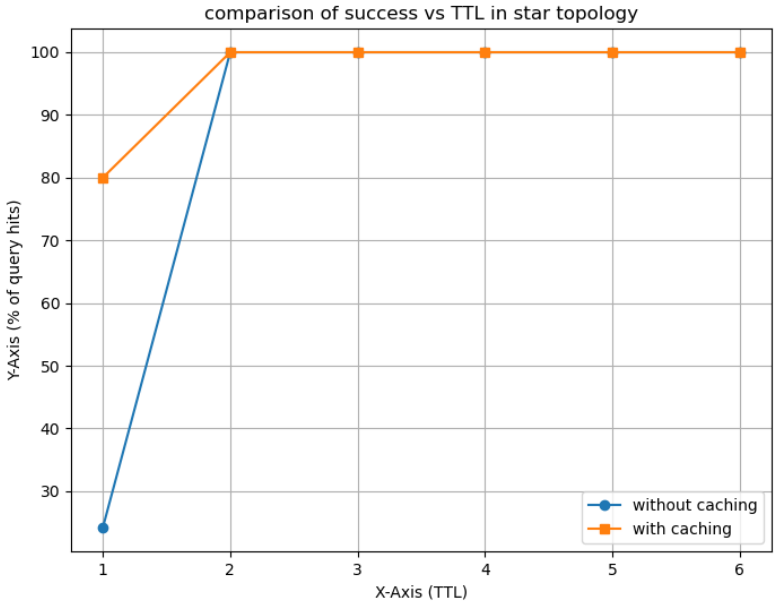


Figure 2: Star Topology

6.2.2 Number of messages as a function of the number of message hops (TTL)



6.2.3 The success rate as a function of the number of message hops (TTL)



6.3 Network Structure 3 (Custom)

6.3.1 Network Diagram

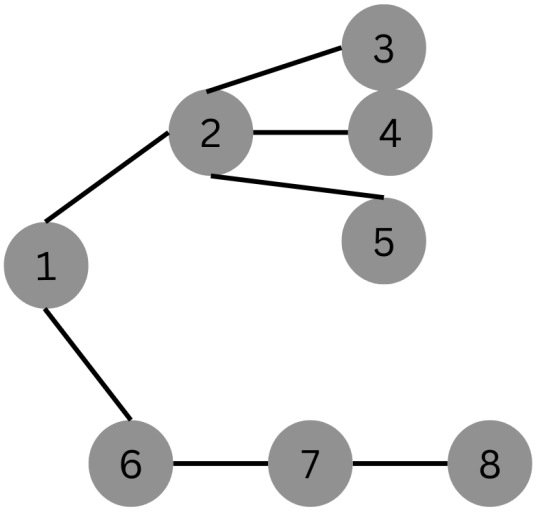
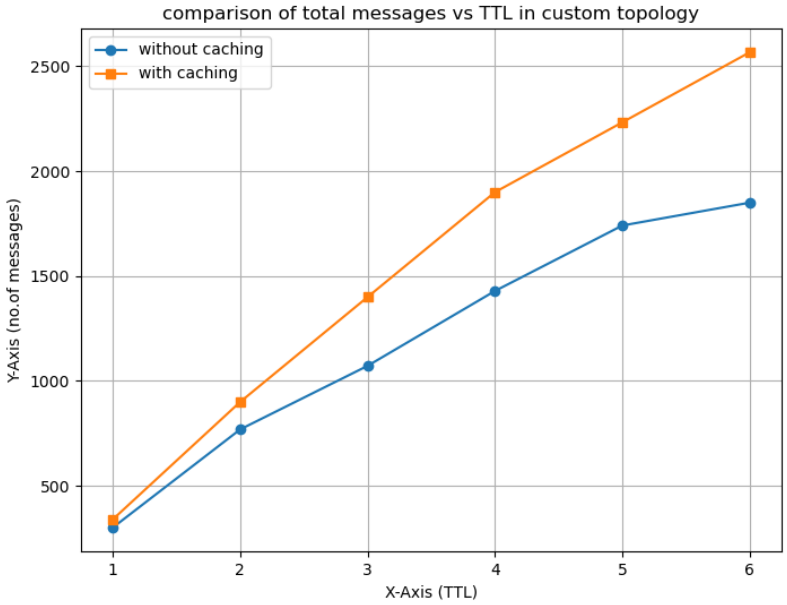
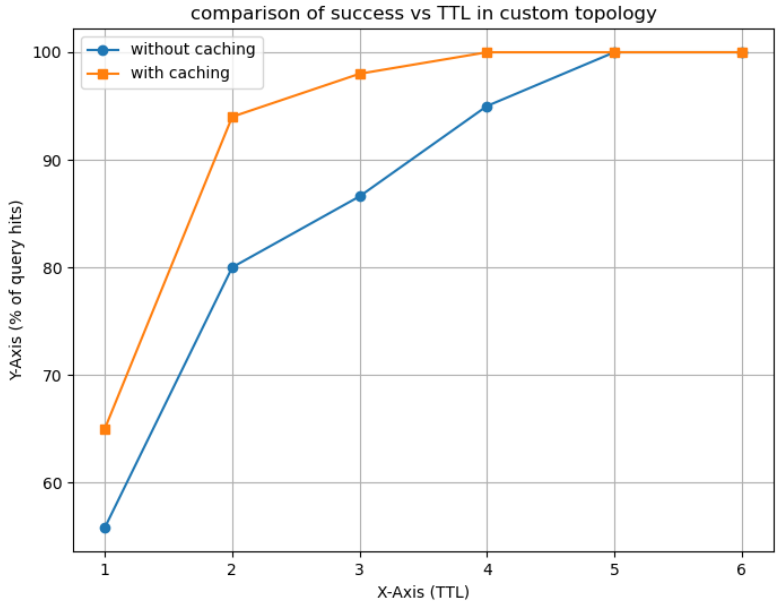


Figure 3: Custom Topology

6.3.2 Number of messages as a function of the number of message hops (TTL)



6.3.3 The success rate as a function of the number of message hops (TTL)



7 Conclusion

As can be observed from our analysis, our system with caching typically takes more number of messages for the same TTL value across most network topologies. However, while this is a minor tradeoff, the reward is huge; there is a higher rate of success in finding a file due to this caching. Furthermore, us ensuring consistent caches essentially means that regardless of

whichever node you retrieve a file from, if that node has previously retrieved that file from another node, then this file will always be up to date at both nodes.

8 Future Work

1. Implement load balancing, so that QueryHit will return only the closest copy of a file depending on network conditions, for multiple copies of a file being detected.
2. Cache Pong and QueryHit messages consistently, so that flooding doesn't have to be performed every single time.
3. Add support for more search parameters in Query messages, apart from just filename.
4. Add support for adding directories of files as a whole to the file which exposes all the files that a node has possession of.

9 Work Split

Kritin: Implemented the file management utilities, and implemented part of the *QueryHit* protocol, as well as the entire *Push* and download protocol. Designed and implemented file consistency on top of the *Push* protocol.

Jayanth: Implemented *Ping*, *Pong*, *Query*, *QueryHit* and designed and implemented the base command line interface (CLI) of the system. Performed analysis of our system.