

# 1 Design Overview

## 1.1 Files

1. `storage.cpp`

This file defines the structure for each storage nodes in the homogeneous structure that powers this distributed file system.

2. `client.cpp`

This class abstracts interactions with the manager for the user.

3. `manager.cpp`

This class is in charge of managing each storage node. Client nodes contact the server before making any GET or PUT requests to client nodes.

## 1.2 Sockets

Each node (client, manager, storage) exposes a unique port that can be connected to by multiple processes. This is the "listen" port. For the manager class, this port is static and can only be configured before runtime. For the other nodes, their listen port is randomly chosen by the OS at runtime and it is their responsibility to make the manager aware of their listening port number. In our case, all nodes are on the same local IP address. However, in the event that nodes are spread out on different machines, the user or daemon starting the storage and client nodes would be responsible for knowing which IP address the manager is on, and the code would be edited to take in a command line input to set the ip address.

## 1.3 Packet Types

There are multiple types of packets in our architecture but each type begins with a `uint_8` header describing the packet type for easy parsing.

The packet types are as follows

1. DISC

used for storage node initialization. This is how the manager discovers the children

2. PUT

used when a client or storage node wants to store a key

3. GET

used when a client wishes to receive a value at a given key

4. ACKPUT

response given by storage node when PUT was successful

5. ACKGET

response given by storage node when GET was successful

6. FAIL

General fail item

7. S\_INIT

Primarily used for storage node initialization. This packet contains group size, members, and the primary node

## 1.4 Storage Node Initialization

Storage nodes are initialized separately from the manager daemon. When a storage node is spawned, it sends a DESC packet to the manager and is added to the pool of available nodes. Each storage node is apart of a group as determined by the manager. Groups are assigned in a round robin fashion to ensure a balanced load across all nodes. In every group there exists one primary node that is the primary communication device for the client. Each node within a group is made aware of its neighbors by an S\_INIT packet from the server.

## 1.5 PUT operation

When a client wants to store a value at a given key into a node, it first asks the manager for the node group responsible for that key. The packet sent to the manager has a field for key and a field for value. The value field is a stringified version of the original vector in which each value is joined by a pipe delimitator. If that key has yet to be tracked, the server uses round robin to find an available group and assigns it to the given key.

The client is then given a group struct with the port number for the primary storage node and a list of it's neighbors for redundancy. This list is never used directly by the client.

A client then sends the same PUT packet to the primary node and is given a ACKPUT response if successful.

In the background, the primary storage node broadcasts that same packet out to all of it's neighbors.

## 1.6 GET operation

When a client desires a value for a given key, it first asks the manager to find the group responsible for that key. If that key does not exist in the manager's housekeeping, a generalized FAIL packet is returned.

If the key does exist, the client is returned a group struct with the port number for the primary storage node and a list of it's neighbors for redundancy. Again, this list is never used directly by the client.

The client then forwards the GET packet to the primary node and is given a ACKGET response if successful.

In this packet, the value field is now non-null. In it is a string delimited by —. The client library splits this string into substrings based around this delimitator and returns a vector to the original caller.

## 1.7 Data Management

1. manager has key group map
  - how its used on put
  - how its used on get
2. each node has a view of what keys it has

## 1.8 Data Redundancy

how data is duplicated across nodes

## 1.9 Node Failure

1. no heart beat and why
2. One node is primary, it has neighbors

Two discovery cases

1. Client discovers node died
2. Node discovers neighbor died

In both cases, a packet of type NODE\_FAILURE is sent to the manager.

# 2 Results