CSCI 5105 Introduction to Distributed Systems

Programming Assignment 2

(Simple File System Based on Gifford's Quorum Protocol)

Arjun Subrahmanyan (Student ID : 5217513)
Devavrat Khanolkar (Student ID : 5211324)

Introduction

The following is a **design document** that describes the different components of the Simple File System (based on **Gifford's Quorum** protocol) that was implemented in **Java** using **Apache Thrift**.

Starting the system

A simple assumption of the system is that each server that will be a part of it knows the IP and port number of the coordinator service. The coordinator service is first started with the parameters N, N_R, N_W provided to it, via command line arguments. A check for the constraints of Gifford's protocol is made, i.e, we check if:
(1) $N_R + N_W > N$ and
(2) $N_W > N/2$
In case of violation, the process (system) is terminated. Once the coordinator service is started, a separate thread is started for **synch** process (details in section 'Coordinator Service').

When each file server joins the system, parameters like port number and the IP and port number of the coordinator are provided via command line. The file server then notifies the coordinator that it has joined the system, before starting a thread separately which provides a UI that lists the files on that server.

Coordinator Service

The coordinator is a multi-threaded service, which means that when a file server contacts it, a new thread is created which processes the request separately.

1. *join(IP, Port)*
   Once the file server starts, it notifies the coordinator with its own IP and port number. The latter then updates its *nodesList* with this information and also tracks the number of servers connected.

2. *assembleQuorum(IP, port, task)*
   This is a function that is called by the file server which receives a read or a write request from a client. As a first step, the coordinator first checks if (number of servers connected = N). If the condition fails, then the client is notified that **the system is not ready**. Else, the read or write quorum is assembled) based on the request (specified by parameter task), as described below.

   The coordinator first creates a *Request* object, which contains details of the IP and port of the server that sent the request, along with its type, namely, read or write. Each request is given a unique ID (*requestCount*) and added to a process queue. The queue is a *thread-safe* data structure. The head of the queue at any point of time represents the next request to be processed.

   In order to facilitate **concurrent reads** and **sequential writes**, we do the following: if the head of the queue is a read request, then remove it and execute it; if it is a write request, then execute it before removing it from the queue. If a write request follows a read request in the queue, then we must also ensure that it does not begin executing before the read request is done. For this, a counter, *runningReads* is maintained, which represents the number of concurrent reads currently being processed. If the head of the queue is a write request, then it is made to wait until *runningReads* becomes equal to 0.

   Based on *task* and the parameters N_R, N_W, a quorum is assembled by randomly selecting that many number of servers from *nodesList*. This quorum, along with the request ID, is returned to the file server that contacted the coordinator with the *corresponding request*.

3. *done(IP, port, task, id)*
   This is a function called by the file server, once it finishes executing the read or write request. If the request was a read, then the counter *runningReads* is decremented, while for a write request, the queue is updated by removing it off (as described before).

4. *synch()*
   This is a background process that is invoked every 10 seconds. The actual synch is not carried out if the number of servers is not equal to N. As part of *synch*, the coordinator obtains the file list, along with their version numbers, from each of the file servers. For each file that exists in the entire system, the latest version number for each is obtained and then the changes are propagated to each of the server, by invoking the *writeAux* method on each of them(described in the next section).

File Server Service

This component handles read and write requests from the client, and stores files that are written, as part of the service. Each file server is multi-threaded to facilitate multiple clients sending requests at the same time to one server.

Note that the coordinator is not responsible for reading/writing the files into the quorum as such. All it does is to assemble a read/write quorum, which it returns back to the server that received the request from the client. The server then contacts each of the servers in the quorum (may include itself) and performs the corresponding operation. This design helps to reduce the load on the coordinator.

1. *write(filename, contents)*
   The file server which receives the write request contacts the coordinator, which returns the write quorum (that possibly includes the former) back. The server then contacts each of the servers in the quorum to obtain the version number of the corresponding file, on that server. If the write operation is for a file that does not exist in the system, then it is written so in the write quorum, with a version number of 1. Else, it is *updated* in the write quorum with a version number equal to 1 more than the maximum version of the file existing in the system. The actual writing of the file is carried out by invoking *writeAux* method. The new version number of the file is updated in the server and *done()* is invoked on the coordinator. The client then gets a message about successful file write.

2. *writeAux(filename, contents)*
   This method writes the file onto the disk, by first creating a folder for each of the file server, with the contents of the file being the filename and the version number itself.

3. *read(filename)*
   The first step in *read()* is similar to *write()*, where the file server that receives the request from the client, first contacts the coordinator. The coordinator then returns back a read quorum, if the system is ready (all servers are connected). The server then contacts each of the servers in the read quorum and obtains the version number of the corresponding file on that server. It then contacts the server with the maximum version number and returns the contents of the file (via *readAux()*) back to the client. If the file does not exist, then we return the message "File not found". Once this read operation is complete, then the file server invokes *done()* on the coordinator, which as described previously, decrements the count of *runningReads*.

4. *readAux(filename)*
   This method opens the corresponding file from the disk (if it exists) and returns back the contents to *read()*, which is then returned to the client. Appropriate error message is returned if the file does not exist.

[**Each server maintains a map *versions*, that is a mapping between a filename and its current version on the server. The following methods work on this map *versions***]

5. *updateVersion(filename, version)*
   The file's version number is updated in *versions*.

6. *getVersion()*
   The file's version number is returned, if it exists in the server, else a 0 is returned indicating absence of the file.

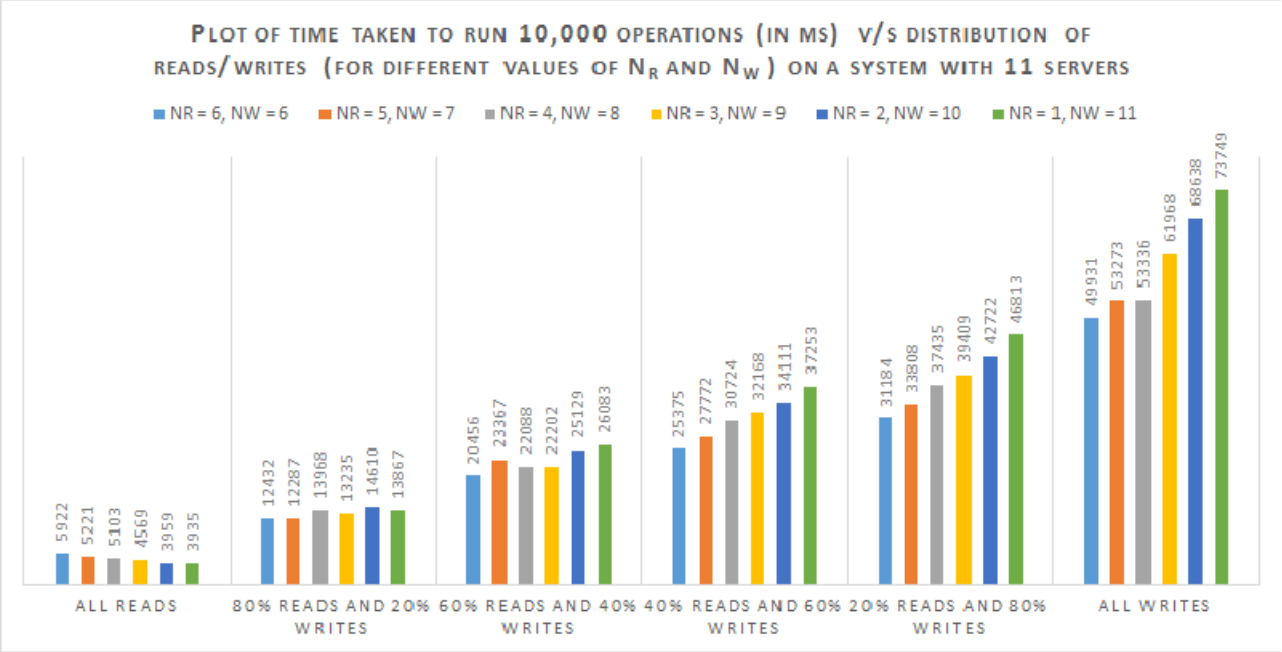
7. *getVersionMap()*
   This is a method that returns the *versions* map. Invoked by coordinator while performing *synch()*, in order to gather entire information of all the files in the system.

Performance Evaluation

The performance of the system was evaluated for different combinations of N_R and N_W. The experiment was performed with **Ubuntu 14.10** operating system and on **localhost environment**. A total of **11** servers were started along with a coordinator. For each value of N_R and N_W, 10000 read/write requests were given with varying distribution of the two operations. The requests were sent via **5** clients, with each of them responsible for **2000** requests.

The following is one plot describing the results of the experiment:

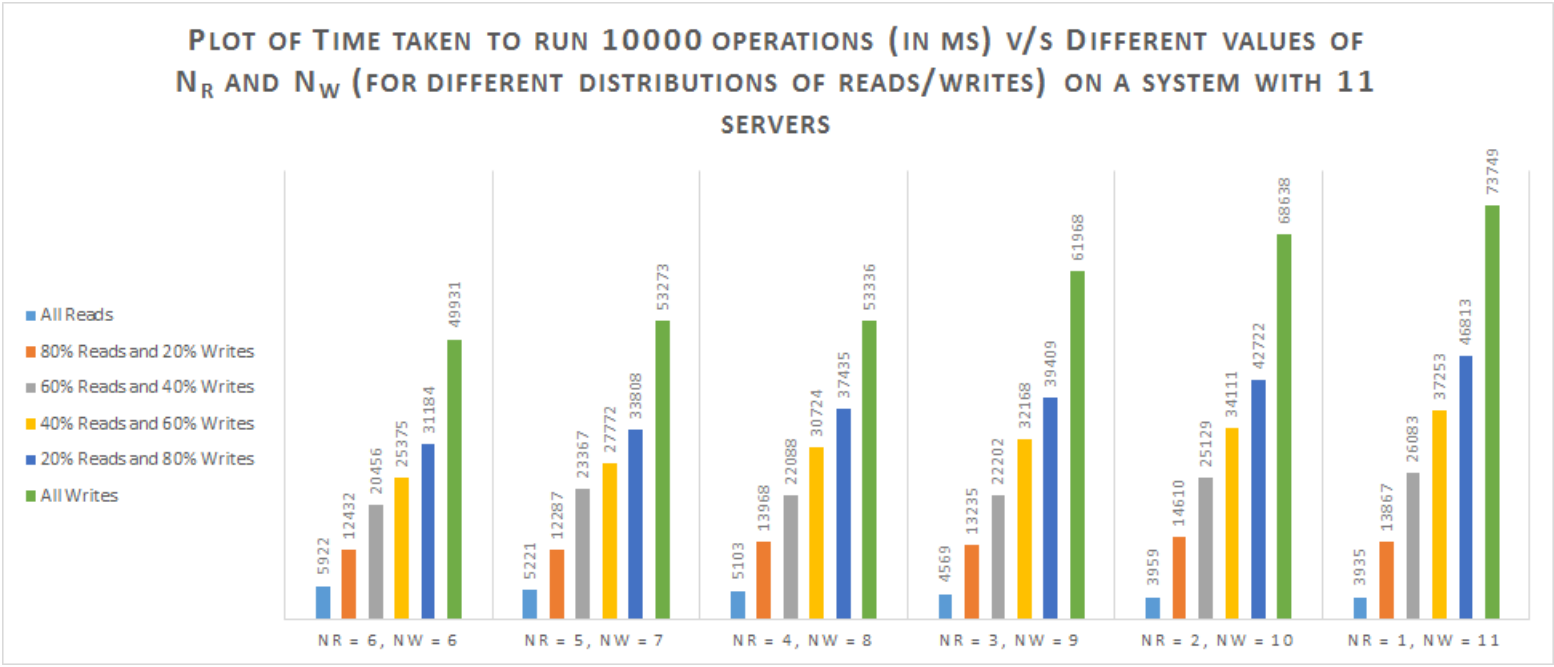
Figure 1:



The above figure tells us that as we decrease N_R, read-heavy operations (like 'All Reads') take lesser time to execute. This is as a result of decreasing number of servers to contact during a read operation, which speeds up the performance of read operation.

Likewise, a decrease in N_R and an increase in N_W implies that the number of servers to write a file into increases, and thus the runtime of a write operation increases. Thus, the time for write-heavy operations (like 'All Writes') increases as we increase N_W.

Figure 2:



Another view of the Figure 1, is shown above, where the reference now becomes the values of N_R and N_W; and for varying distribution of read and write operations, the time taken for all the 10000 operations to complete is shown.

For a fixed value of N_W, we see that the runtime of these operations increases from a read-heavy to a write-heavy distribution since writes are sequential and reads are made concurrent.