CS/CE/TE 6378: Project III

Instructor: Ravi Prakash

Assigned on: April 12, 2011
Due date and time: April 28, 2011, 11:59 pm

1 Requirements

1. This is an individual project and you are expected to demonstrate its operation to the instructor and/or the TA.
2. Source code must be in the C /C++ /Java programming language.
3. The program must run on UTD lab machines (`net01, net02, ....`).

2 Client-Server Model

In this project (similar to project I and II), you are expected to use *client-server* model of computing. You may require the knowledge of thread and/or socket programming and its APIs of the language you choose. Also, make sure that processes (server/client) are running on different machines (`netXX`).

3 Description

This project is extension of project II, in which you will implement two new features to the distributed file system –

1. A new file-server may be added dynamically in order to share workload of the existing file-servers. For example, if each of the existing file-servers S_1, S_2 and S_3 (see Figure 1) is hosting four files then, after introducing a new file-server S_4 to the system, one file from each of S_1, S_2 and S_3 can be migrated to S_4. So, each file-server would host only three files instead of four while preserving the total number of files in the file-system and maintaining the replication level at three. A file-server may initiate the migration of a file by sending *File Offload* request to M-server.

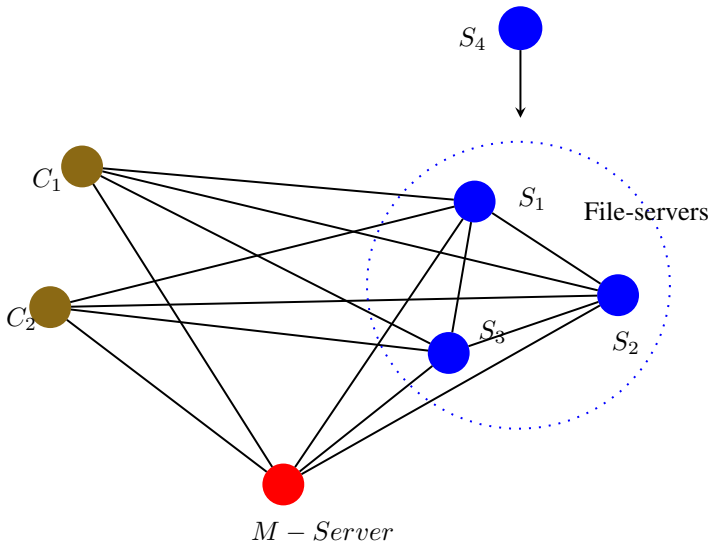

Figure 1: Network Topology

2. M-server provides services based upon meta data populated by periodic updates from the file-servers. However, M-server may crash at some point of the time and lose its current meta data. A new M-server (or the same M-server) may come up after the crash of original M-server on the same address (IP, port). The file-system should be able to handle M-server failure and it should be able to restore connectivity with M-server again. Also, since M-server has lost its old data (meta data), it should re-populate meta data based on periodic updates from the file-serves.

You can assume that M-server does not crash while a new file-server is being added into the system.

The M-server must implement following services –

1. It receives periodic updates from the file-servers about their local files' status and updates its own meta data.
2. It has knowledge of all servers, clients and files hosted by the servers.
3. It keeps track of files along with current (to the best of its knowledge) location and status of files. The possible status of a file could be:
    a. CLOSED : The file is closed and not being used by any client.
    b. R_OPEN: The file is open in read-mode by some client(s).
    c. W_OPEN: The file is open in write-mode by some client.
    d. IN_TRAN: The file is being distributed from one file-server to some other file-server(s).
4. The M-server also keeps track of multiple users (clients) of a file (in case of multiple read).
5. Upon receiving a request (Create|Read|Write) for permission from a client for a particular file, it should reply with either REJECT or GRANT permission to perform the corresponding operation. If it grants the permission then the following is done:
    (a) If a client is allowed to open a file in the read mode, the M-server randomly selects one of the file-servers where the read is to be performed and informs the client of its selection.
    (b) If a client is allowed to open a file in the write mode, the M-server communicates to the client the identities of all the file-servers maintaining replicas of that file.
6. Upon receiving a *File Offload* request from a file-server, it should reply with GRANT if the file is in CLOSED status. Otherwise, it should delay the GRANT until the file's status becomes CLOSED.
7. On detecting a crashed client, if any, it instructs the server(s), on behalf of the crashed client, to close all files opened by the crashed client, and updates its local meta data.
8. Upon request, it sends information to a file-server about presence of other file-servers and files on those.
9. It should not give GRANT permission for Read/Write to a file which is in IN_TRAN state.

The desired operations by a client are as follows:

1. A client must obtain GRANT permission from M-server before processing each request. In case the permission is granted, the client can use the SERVER location address (provided by M-server) for further communication. If the permission is denied (REJECT) or M-server is not available, it should abort the request.
2. It may send CREATE a file request, READ from a file request or WRITE to a file request to the SERVER upon having GRANT from M-server for corresponding request.
3. It performs READ/WRITE request in following three steps –
    I. Opens the file in read/write mode and sets the status of the file to R_OPEN/W_OPEN.
    II. Transfers data between client and server(s) and read/write the content from/to the given file.
    III. Closes the file and sets the file's status to CLOSED.

So, any READ/WRITE operation results in three sets of request-response communications between client and server.

While M-server is not available, client and file-server may continue file operations if the client already has GRANT permission from M-server.

The desired operations by a file-server are as follows:

1. It locally populates meta data of files that it hosts, which also contains the current status of files (CLOSED|W_OPEN|R_OPEN|IN_TRAN).
2. It must send periodic updates of its meta data to M-server.
3. Upon request from a client, it must be able to CREATE a new file, WRITE and READ from a local file.
4. In case of CREATE, *all* copies of the file, across the servers, should be updated consistently.
5. In case of WRITE, since it has three operations, each operation must be completed on *all* servers before next.
6. It must reply either SUCCESS or FAILURE to the requests from client along with requested data (if any).
7. It may request permission for *File-offload* to M-server for a file. Upon receiving GRANT, It may initiate off-loading and transfer the file to other file-server(s).

As part of this project you have to determine how a server, on receiving the WRITE request, communicates with the other servers to get the same write performed on all the copies of the file. Your program must support the creation of new files, writes to the end of files, reads and writes at specific offsets from file beginning. It must also report an error if an attempt is made to READ that failed at the requested server for some reason. An error must be reported if there is an attempt to WRITE and at least one of the servers, who hosts the file, is unable to write to it. While creating a new file, you may create a local copy to all currently available file-servers. A client may crash during a request processing, for example, just after completion of step I or step II of READ/WRITE operation. Your file-servers must have some user interface to initiate off-loading of a particular file to *some*, *all* or *none* of the other file-servers. After transferring the file, the local copy should be deleted. Your program must display informative log messages on console.

4 Submission Information

The submission should be through eLearning in the form of an archive consisting of:

1. File(s) containing the source code.
2. The README file, which describes how to run your program.

DO NOT submit unnecessary files.