Distributed File System

Author: Gihwan Kwon, Haram Kwon

Documentation

NOTE:

- Both FileClient and FileServer programs do not require “rmiregistry port# &” command before starting the program.
    - The main programs start the registry.
- The FileServer rmi naming
    - “rmi://” + server Ip + “:” + port# + “/fileserver”
    - ex) rmi://cssmpi1.uwb.edu:35000/fileserver
- The FileClient rmi naming
    - “rmi://” + client Ip + “:” + port# + “/fileclient”
    - ex) rmi://cssmpi1.uwb.edu:35000/fileclient

FileClient.java

- The FileClient program prompts user to decide whether to continue or quit the program (extra-credit). If user owns the write permission, it will upload the latest changes to the server before closing the program.
- The name of cache file (/tmp/<username>.txt) is dynamic in that the program will create a cache file with the current system’s user.name.
    - If user id is johh1111, the program create cache file name /tmp/johh1111.txt
    - If user id is jose2222, the program create cache file name /tmp/jose2222.txt
- The state transitions of the cache file are implemented with the switch statement and the enum data structure.
- The functionalities such as I/O to the local files and changing access mode to the cached file were implemented with the modularized package called “java.nio.file.*”
- Otherwise, the client program does not have unique implementation that is different from the original specification.

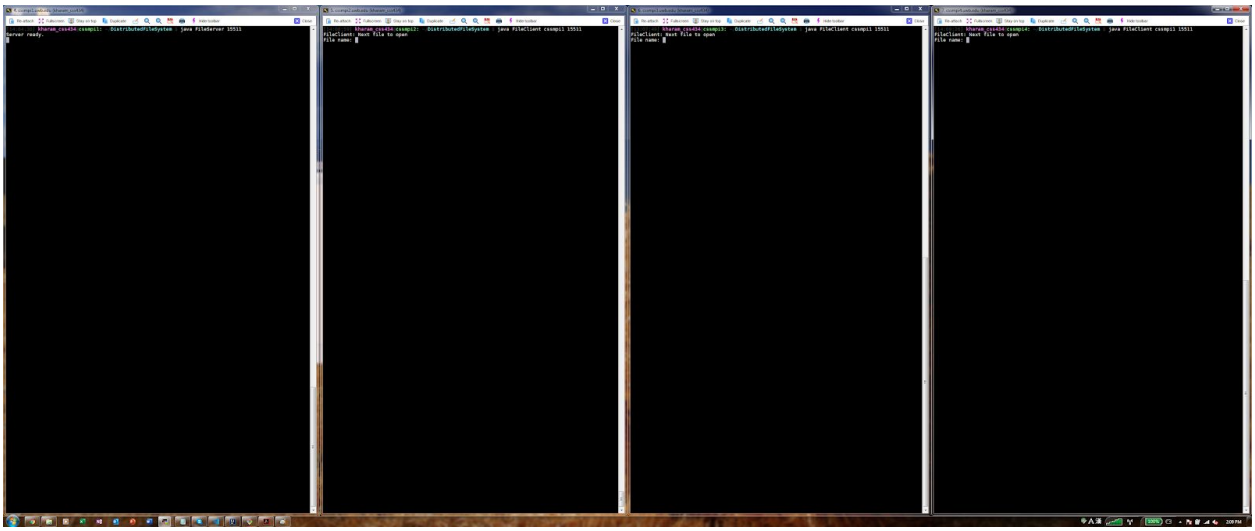
FileServer.java

- The server invokes the monitor to handle the multiple thread.
- All the errors are handled by Exception
- Makes a private file function to track the state of each file.

    - The server can handle multiple read while handling a write.
- Start registry function to start the rmiregistry.
    - To run the program, we don't need to use rmiregistry <port> anymore.
- Invoke switch statement to control the state.

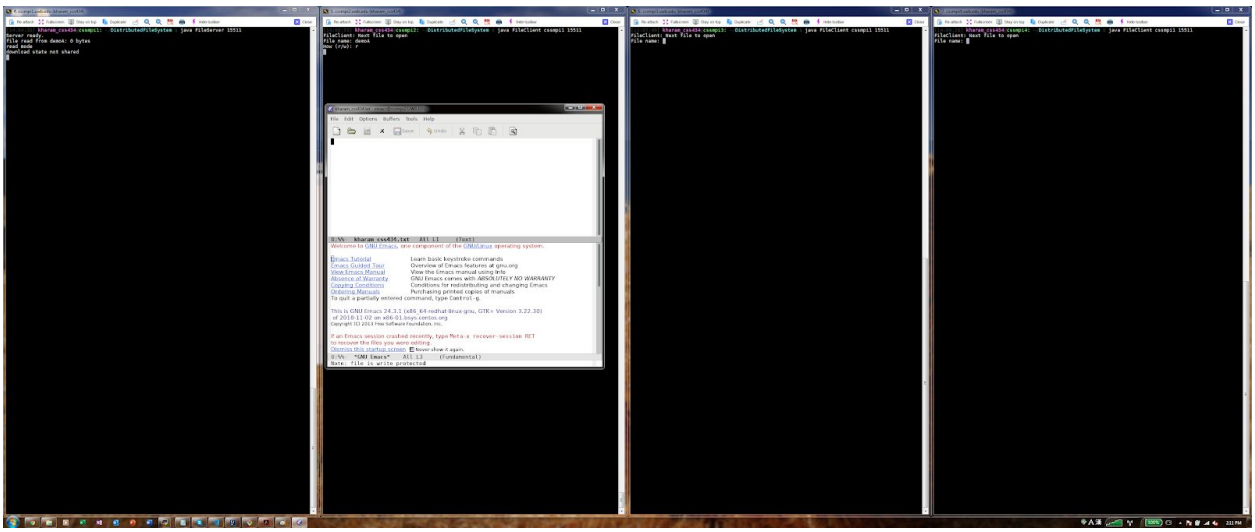
Execution Output

Setup Test environment



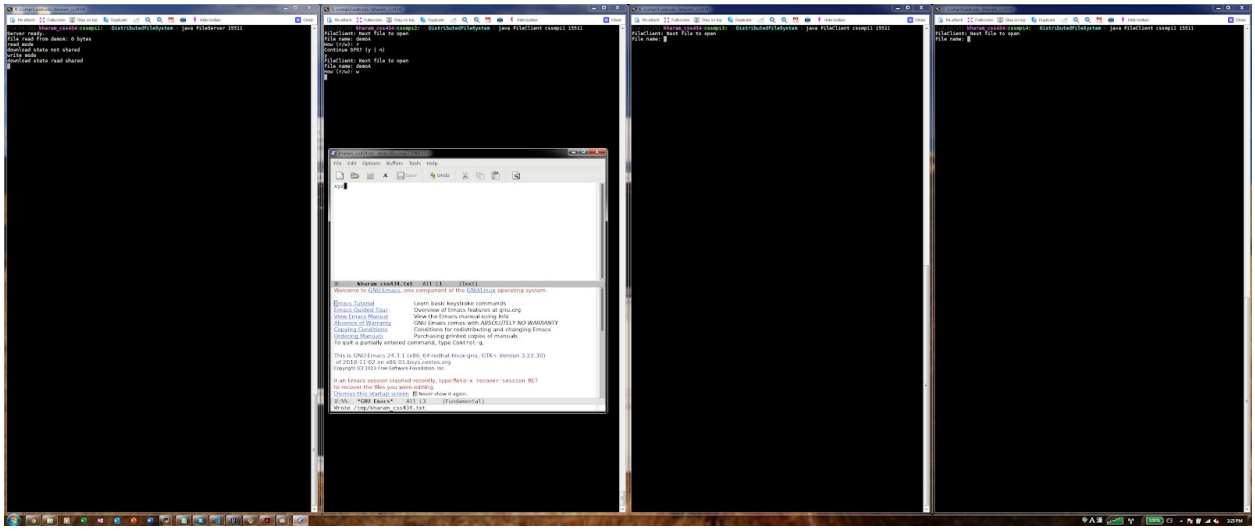
File read test:

- Read empty from demoA at cssmpi2

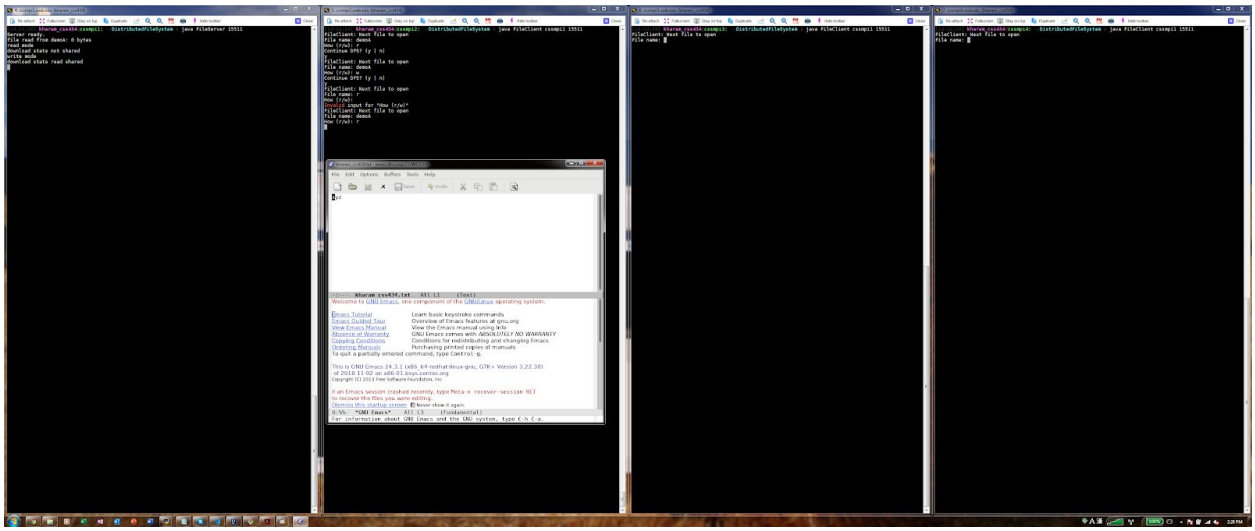


File write test:

- write xyz to demoA at cssmpi2

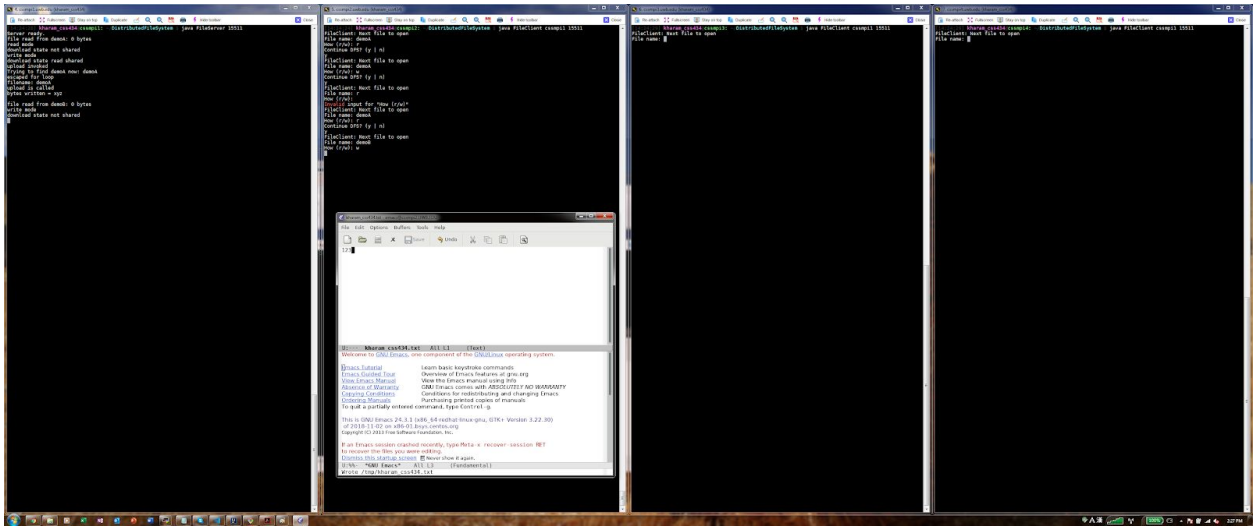


- read xyz from demoA at cssmpi2

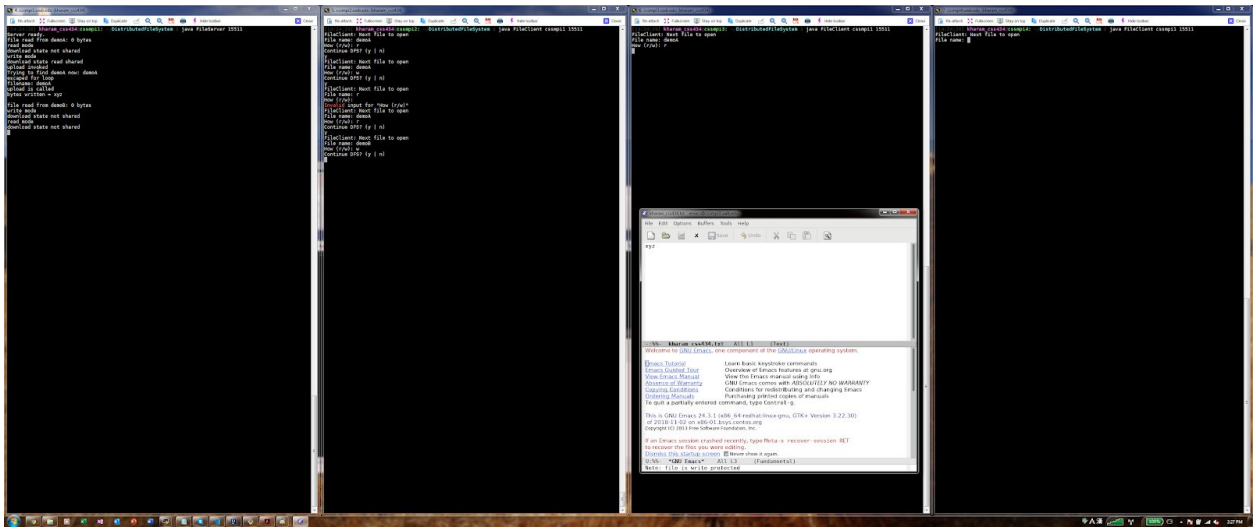


File replacement test:

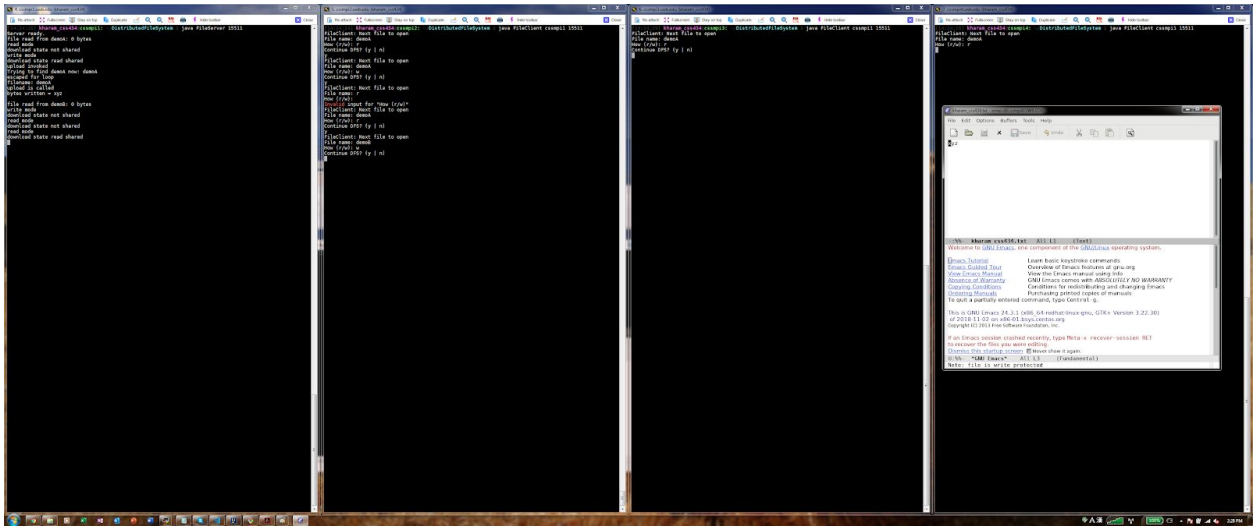
- write 123 to demoB at cssmpi2



- read xyz demoA at cssmpi3

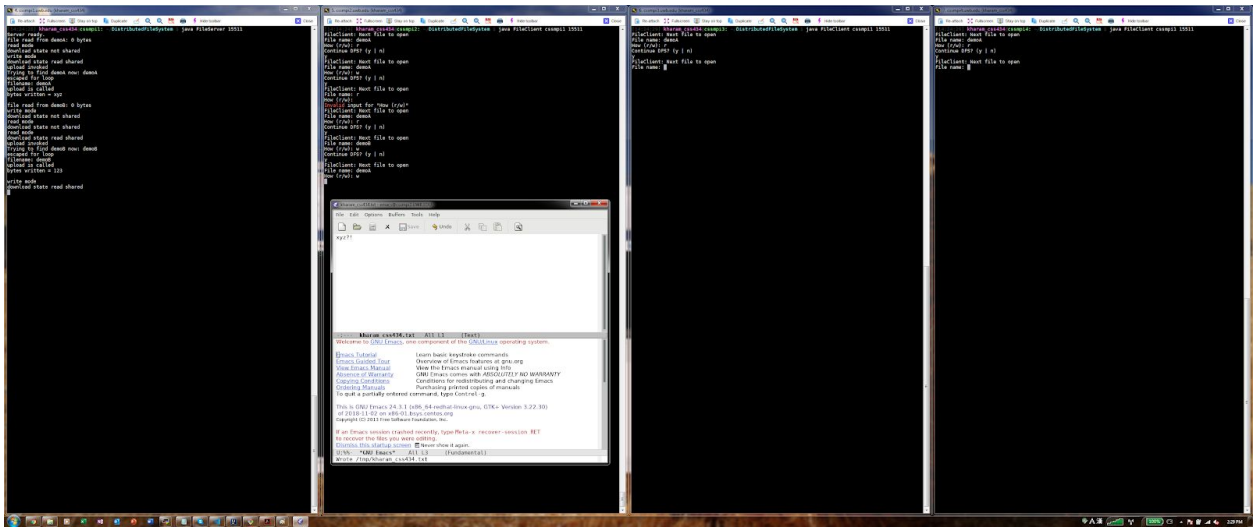


- read xyz demoA at cssmpi4

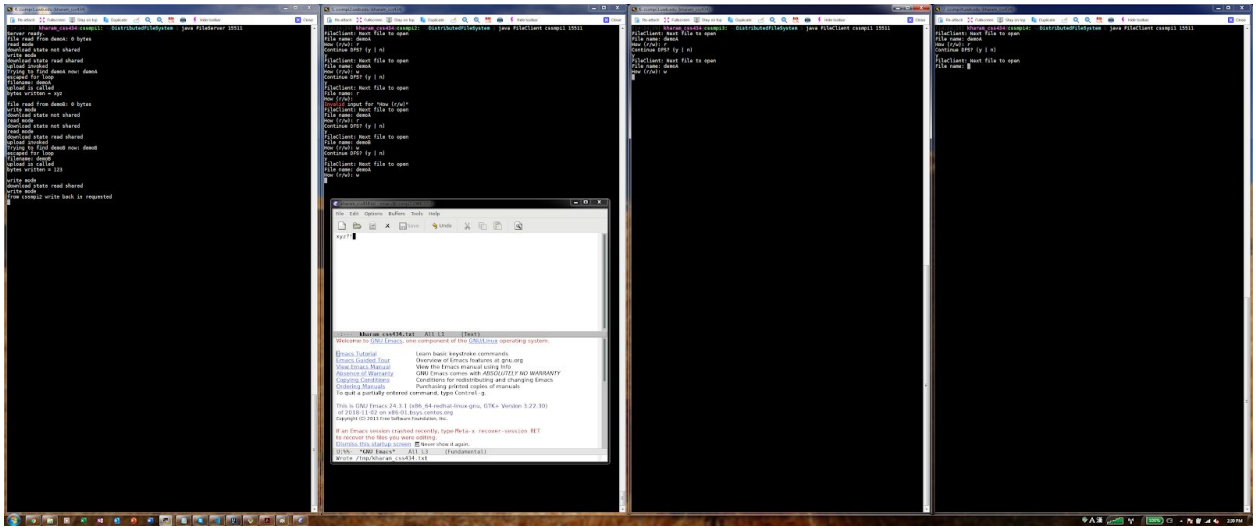


File writeback test:

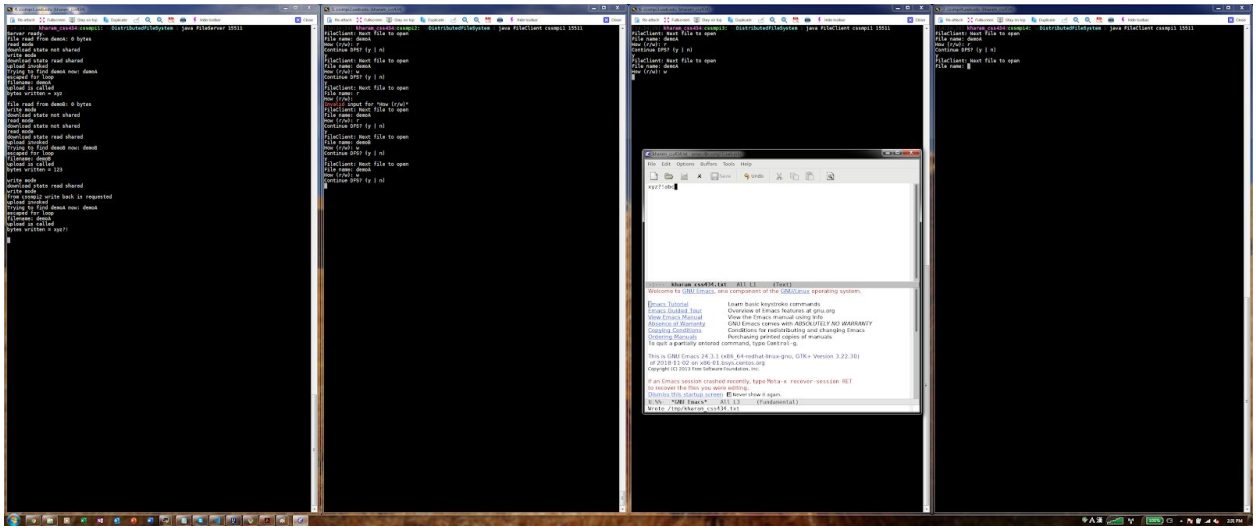
- write xyz?! to demoA at cssmpi2



- write to demoA at cssmpi3 & keep emacs open at cssmpi2

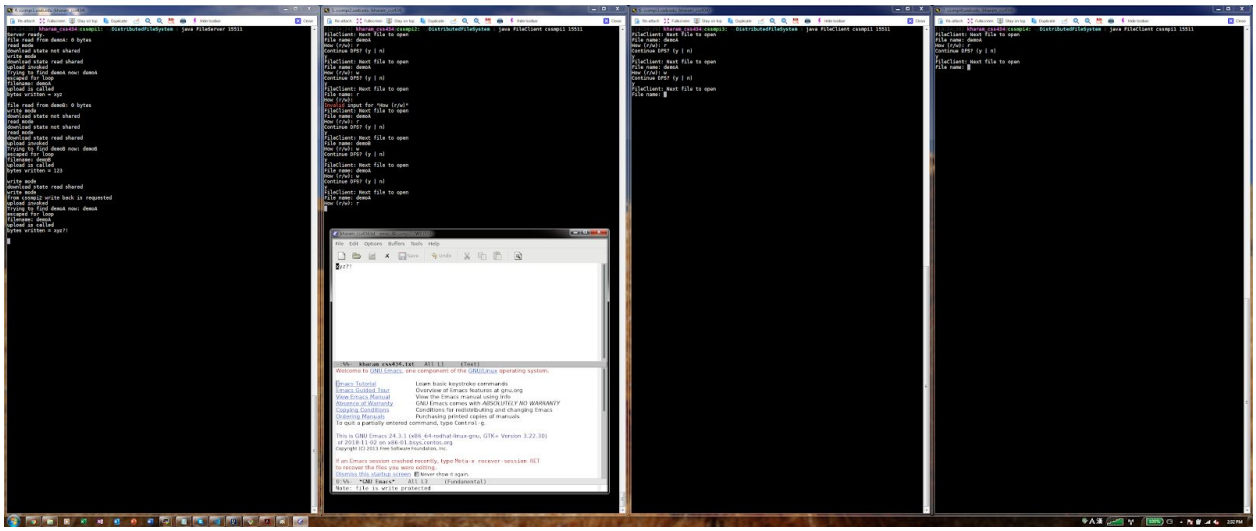


- close emacs at cssmpi2 & write xyz?!abc to demoA at cssmpi3

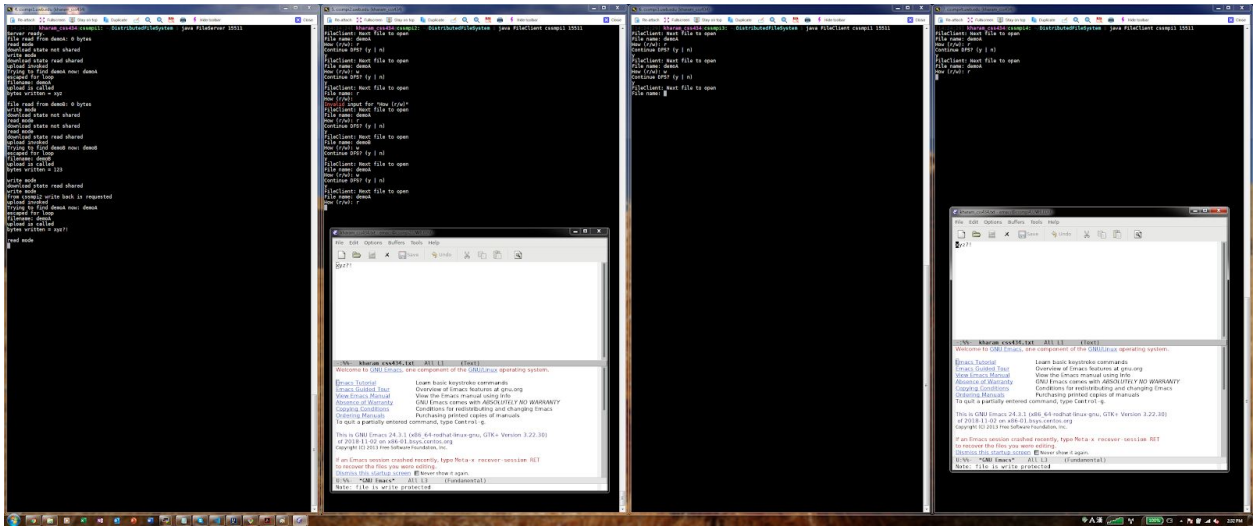


Session semantics read test:

- read xyz?! from demoA at cssmpi2

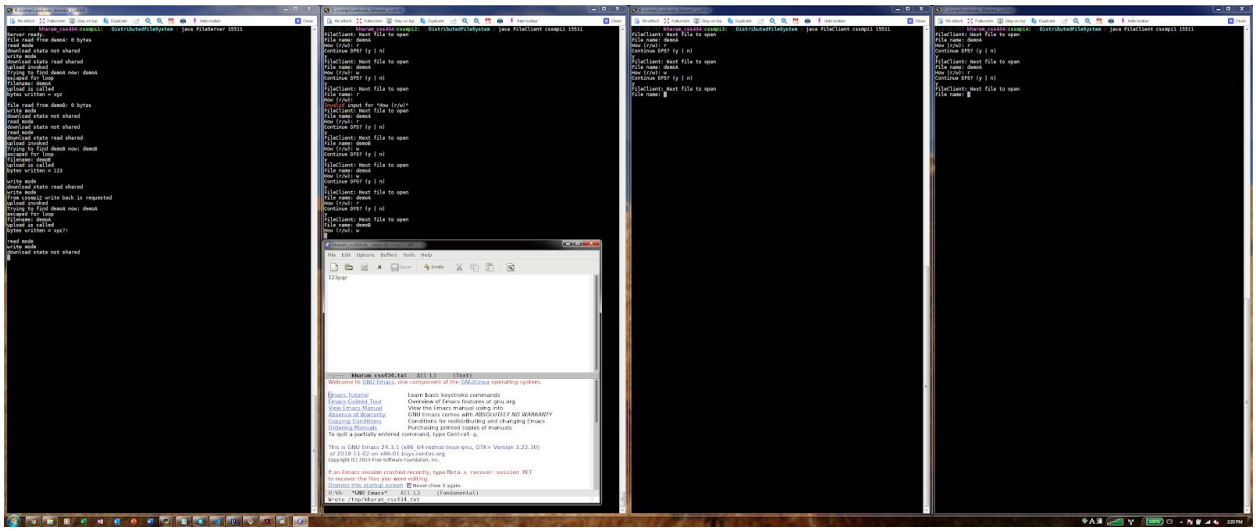


- readxyz?! from demoA at cssmpi4

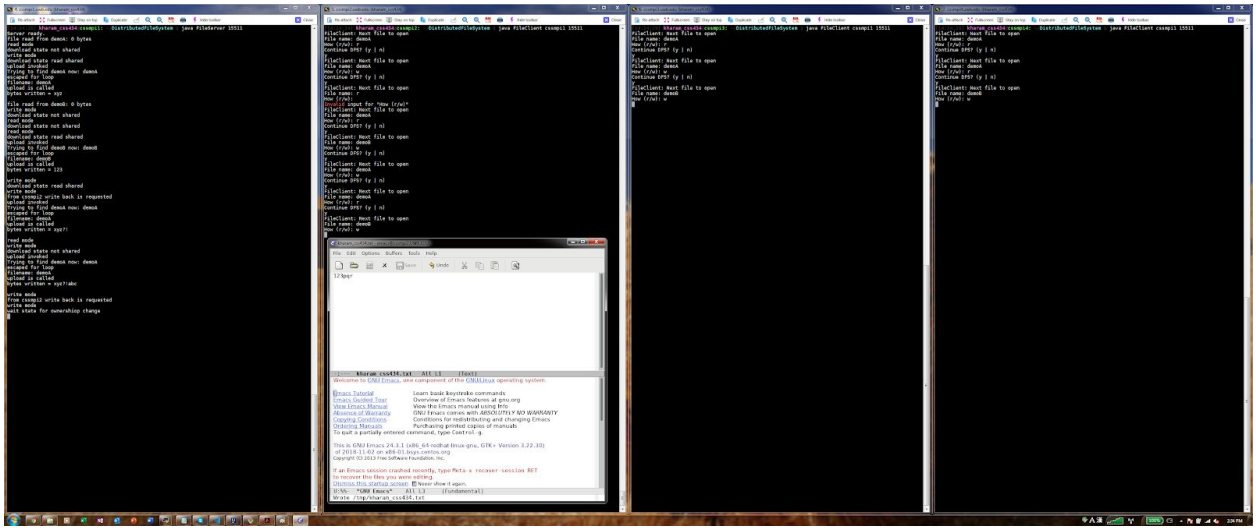


Multiple write test:

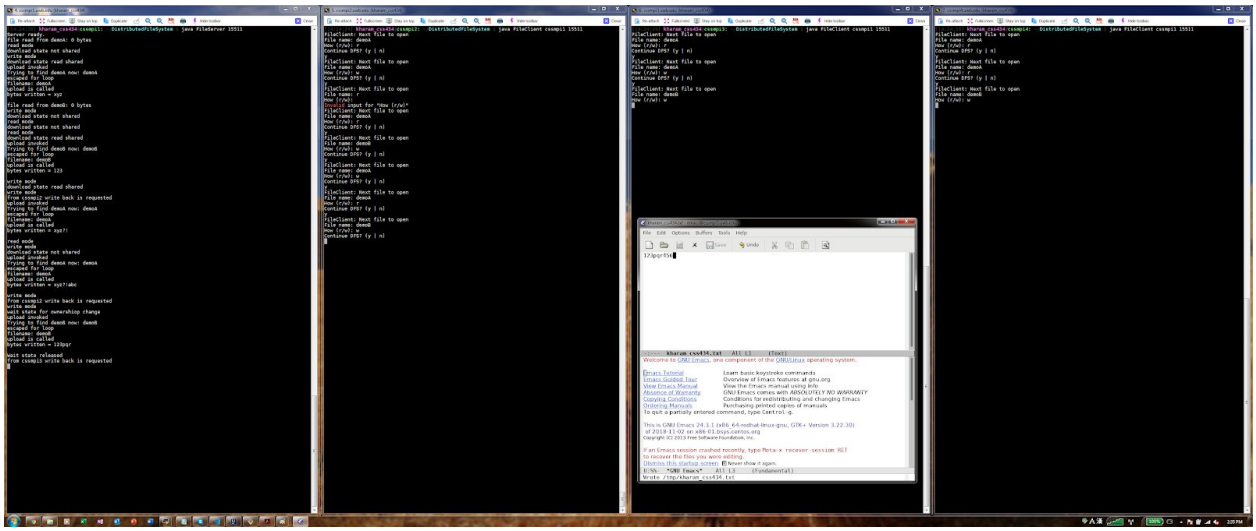
- write 123pqr to demoB at cssmpi2



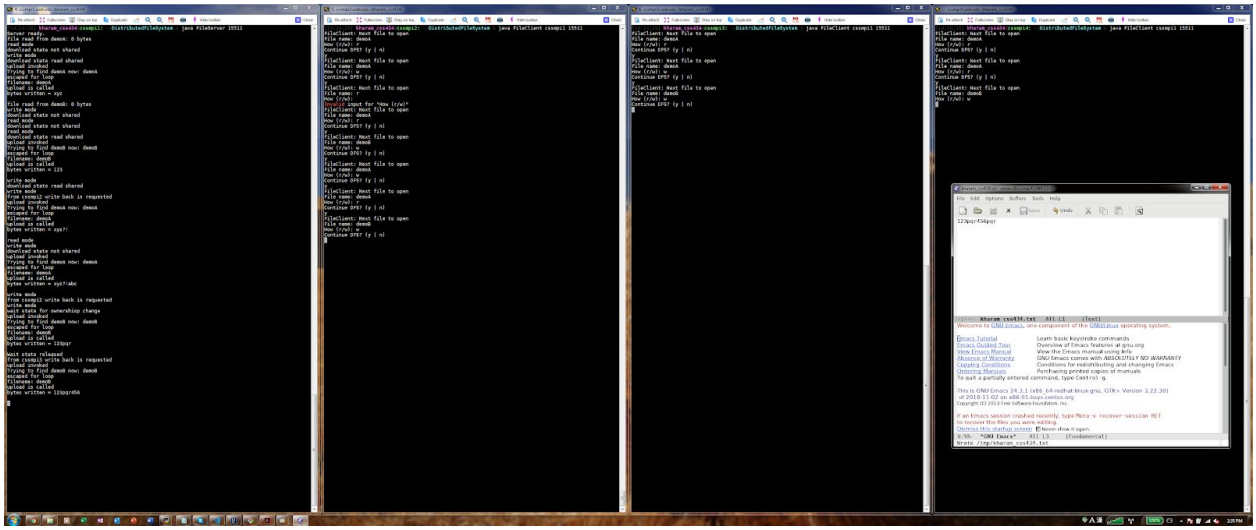
- keep emacs open at cssmpi2



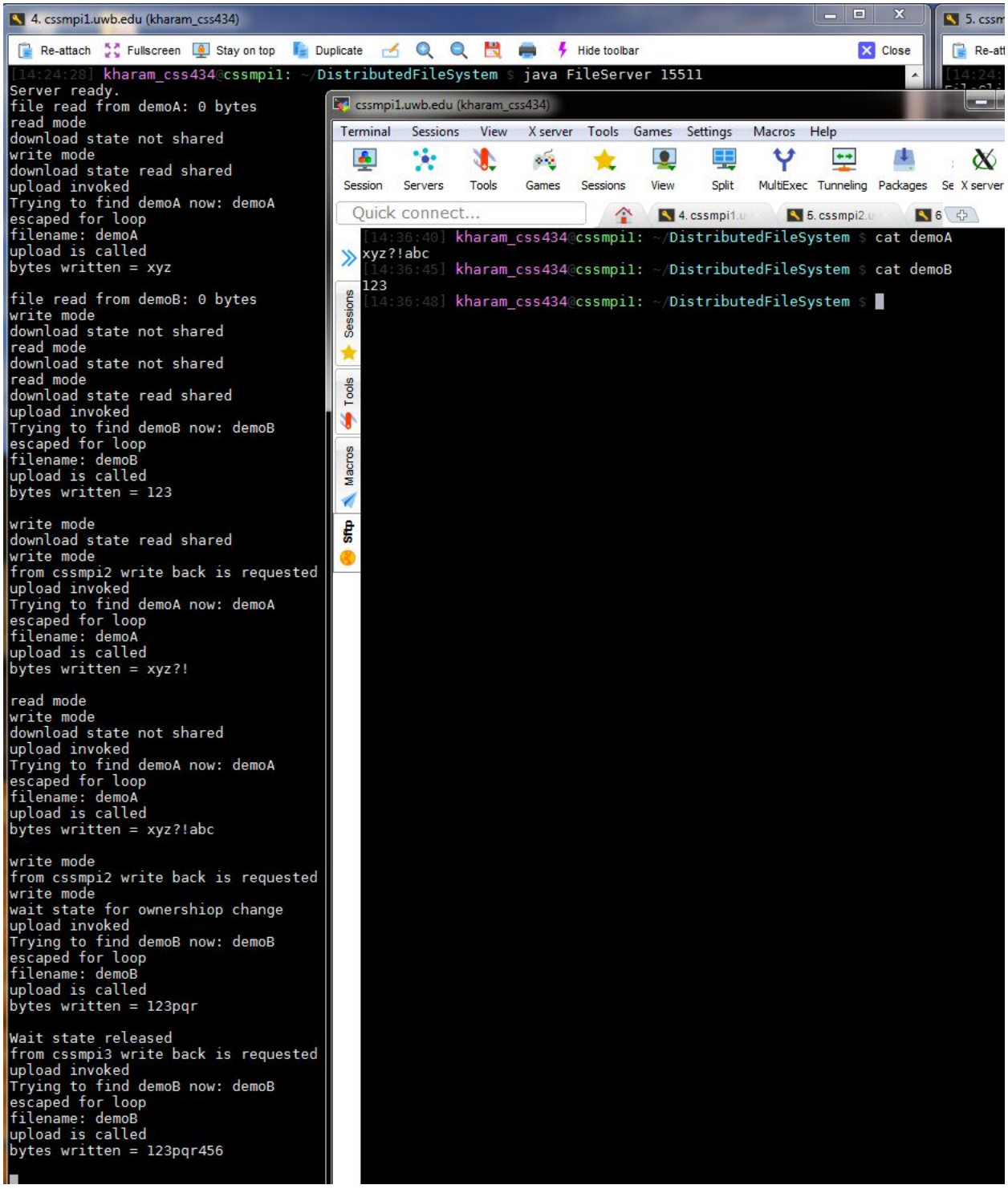
- close emacs at cssmpi2 & write 123pqr456 to demoB at cssmpi3



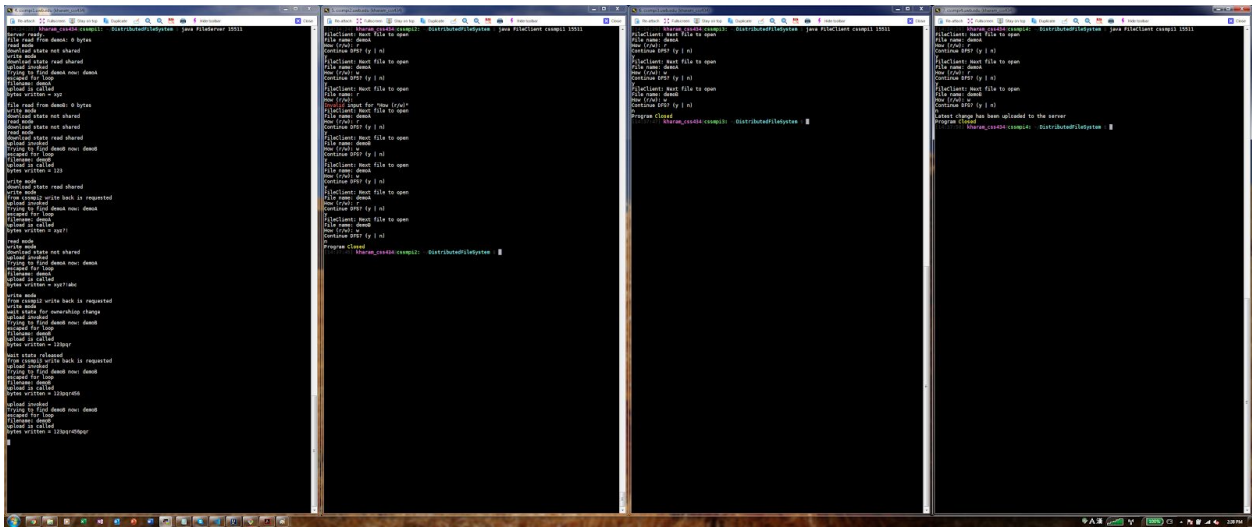
- close emacs at cssmpi3 & write 123pqr456abc to demoB at cssmpi4



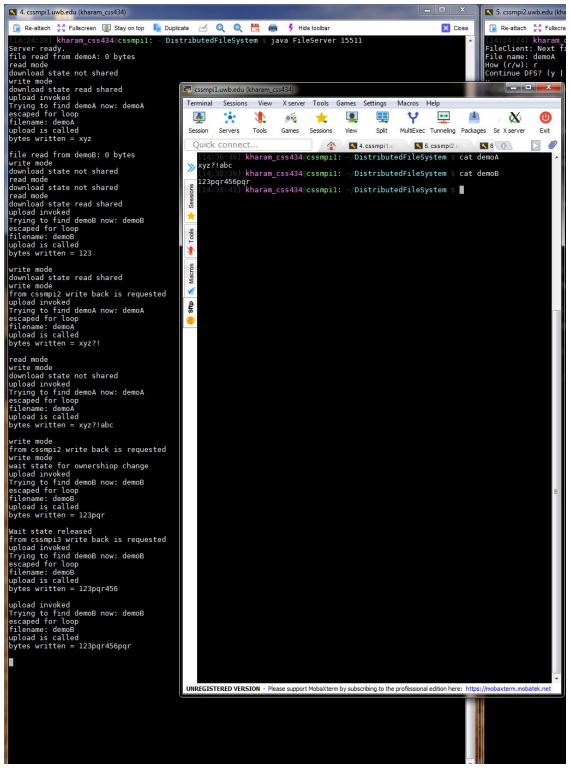
- close emacs at cssmpi4 & quit cssmpi1, 2, 3, 4 (check demoA and demoB with cat demoA ==”xyz?abc” demoB==”123”)



- Our version of closing program. Only cssmpi4 upload its latest local version to the server's disk **(extra-credit)**



- demoA & demoB after closing the program in our version (extra-credit)



Discussion

- Functional Improvements
    - The system should be able to handle multiple copies of the local cached files. In order to implement such functionality, the server program should be able to keep track of the users cache status in a more efficient way. For client program, it should have such functionality that user does not have to immediately write or read file after the request.
    - For the write ownership change state, we can ask the user to wait for write and merely access the file and read only.
- Performance improvements
    - Since the monitor has some overhead, compared to semaphore or mutex, we can invoke the semaphore in java to increase the performance.
    - Use a Map to manage the files instead of vector, we can reduce the time complexity of file look up from O(n) to O(1)

Source Code

FileClient.java

```
import java.rmi.server.UnicastRemoteObject;

public class FileClient extends UnicastRemoteObject implements ClientInterface {

   private File file;
   private ServerInterface serverObject;
   private String localHost;
   private String cacheFile;
   private String fileName;
   private String accessMode;
   private State currentState;

   public FileClient(ServerInterface serverObject, String localHost)
           throws Exception {

       // initialize the local cache file path: /tmp/username.txt)
       this.cacheFile = "/tmp/" + System.getProperty("user.name") + ".txt";
       // System.out.println(System.getProperty("user.name"));

       // initialize the local cache file with the initialized path
       this.file = new File(this.cacheFile);

       // creating new file if it does not exist in the path
       if (!this.file.exists()) {
           this.file.createNewFile();
           this.file.setWritable(true, true); // chmod 600
       }

       // init serverObject for rmi calls
       this.serverObject = serverObject;

       // init the name of local host
```

```
        this.localHost = localHost;

        // init current state (no file)
        this.currentState = State.INVALID;

        // no file in the cache, so no name
        this.fileName = "";
    }

    public static void main(String[] args) {

        // Checking arguments
        int port = 0;
        String localHost = "";
        try {

            // arg validation
            if (args.length == 2) {
                // argument[1] = port#
                port = Integer.parseInt(args[1]);
                if (port < 5001 || port > 65535)
                    throw new Exception();
            } else {
                throw new Exception();
            }

            // name of local host "cssmpi#"
            localHost = InetAddress.getLocalHost().getHostName();
        } catch (Exception e) {
            System.err.println("usage: java Client serverIp port");
            System.exit(-1);
        }

        // argument[0] = server ip
        String serverIp = args[0];

        try {
```

```
            // Find server object
            ServerInterface serverObject = (ServerInterface) Naming
                    .lookup("rmi://" + serverIp + ":" + port + "/fileserver");

            // start rmi registry for client object
            startRegistry(port);
            FileClient client = new FileClient(serverObject, localHost);
            Naming.rebind("rmi://localhost:" + port + "/fileclient", client);

            // start the program
            client.userPrompt();

        } catch (Exception e) {
            e.printStackTrace();
            System.exit(-1);
        }
    }

    // start rmi registry given by Dr. Fukuda
    private static void startRegistry(int port) throws RemoteException {
        try {
            Registry registry = LocateRegistry.getRegistry(port);
            registry.list();
        } catch (RemoteException e) {
            Registry registry = LocateRegistry.createRegistry(port);
        }
    }

    // interact with the user
    public void userPrompt() throws Exception {
        Scanner input = new Scanner(System.in);
        String fileName;
        String mode;

        while (true) {

            // Prompt user for inputs
```

```
System.out.println("FileClient: Next file to open");

// receive the name of requesting file
System.out.print("File name: ");
fileName = input.nextLine();

// receiving the mode
System.out.print("How (r/w): ");
mode = input.nextLine();

// input mode is neither "r" nor "w" re-prompt
if (!mode.equals("r") && !mode.equals("w")) {
    System.out.println("Invalid input for \"How (r/w)\"");
    continue;
}

// if cannot open the file, or any error occurs, reprompt
if (!this.openFile(fileName, mode)) {
    continue;
}

// open requested file on the Emacs according to the mode
openEmacs();

// complete the session after read/write operation
completeSession();

// prompt user for continuation
System.out.println("Continue DFS? (y | n)");
if (input.nextLine().toLowerCase().startsWith("n")) {
    if (this.currentState == State.WRITE_OWNED) {
        System.out.println(
                "Latest change has been uploaded to the server");
        this.saveStateToServer();
    }
    System.out.println("Program Closed");
    System.exit(0);
```

```
        }
    }
}

private void saveStateToServer() throws Exception {
    FileContents currentContent =
            new FileContents(Files.readAllBytes(this.file.toPath()));
    this.serverObject.upload(this.localHost, this.fileName, currentContent);
}

private boolean openFile(String fileName, String mode) {
    try {
        // Before file Replacement happens,

        // if local cache is not requested file,
        if (!this.fileName.equals(fileName)) {
            // files don't match,
            // upload current file content to server if state is
            // writeowned
            if (this.currentState == State.WRITE_OWNED) {
                FileContents currentContent =
                        new FileContents(
                                Files.readAllBytes(this.file.toPath()));
                this.serverObject.upload(
                        this.localHost, this.fileName, currentContent);

                // set state to invalid
                // so client can download desired file from server
                this.currentState = State.INVALID;
            }
        }

        // check state of cache
        // to determine if client downloads server file or not
        switch (this.currentState) {

            case INVALID:
```

```
            // download requested file no matter what
            if (!this.downloadRequestedFile(fileName, mode)) {
                return false;
            }

            break;

        case READ_SHARED:

            // cache exists
            if (this.fileName.equals(fileName)) {

                // requested writing mode
                if (mode.equals("w")) {

                    // read shared state,
                    // re-download and make it writable
                    if (!this.downloadRequestedFile(fileName, mode)) {
                        return false;
                    }

                }

                // otherwise, do nothing just use cache

            } else { // cache does not match requested file

                // should request the requested file
                if (!this.downloadRequestedFile(fileName, mode)) {
                    return false;
                }
            }

            break;

        case WRITE_OWNED:
```

```
                if (!this.fileName.equals(fileName)) {

                    // request different file
                    if (!this.downloadRequestedFile(fileName, mode)) {
                        return false;
                    }

                } // otherwise, do nothing just use same cache File

                break;

            default:
                System.out.println("Cannot open the requested file");
                // something happend,
                return false;

        }
    } catch (Exception e) {
        e.printStackTrace();
        // something happend
        return false;
    }
    return true;
}

private boolean downloadRequestedFile(String fileName, String mode) {
    try {

        // downalod specified file in mode
        FileContents result =
                this.serverObject.download(this.localHost, fileName, mode);

        // no file exists
        if (result == null) {
            System.out.println("The file does not exist in the server");
            return false;
        }
```

```
            // make the cache file writable so that client program can
            // write file content to the cache file
            this.file.setWritable(true, true); // chmod 600

            // write requested file contents into the cache file.
            FileOutputStream tempFileWriter = new FileOutputStream(this.file);
            tempFileWriter.write(result.get());
            tempFileWriter.close();

            // update the name of the cache file
            this.fileName = fileName;

            if (mode.equals("w")) {
                // already writable mode, do not have to change permission
                this.currentState = State.WRITE_OWNED; // write owned state
                this.accessMode = mode; // access mode = w
            } else {
                this.file.setReadOnly(); // chmod 400
                this.currentState = State.READ_SHARED; // read shared state
                this.accessMode = mode; // access mode = r
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
        return true;
    }

    private void openEmacs() {
        String[] command = new String[]{"emacs", this.cacheFile};
        try {
            Runtime runtime = Runtime.getRuntime();
            Process process = runtime.exec(command);
            process.waitFor();
        } catch (Exception e) {
            e.printStackTrace();
        }
```

```
}

private void completeSession() {
    try {
        // if serever notified to release the ownership,
        if (this.currentState == State.RELEASE_OWNERSHIP) {

            // upload changes to the server
            FileContents currentContent =
                    new FileContents(
                            Files.readAllBytes(this.file.toPath()));
            this.serverObject.upload(
                    this.localHost, this.fileName, currentContent);

            // set state to read_shared
            this.currentState = State.READ_SHARED;
            this.file.setReadOnly(); // chmod 400

        }
    } catch (Exception e) {
        e.printStackTrace();
    }

}

// rmi call for the server to invalidate the cache
public boolean invalidate() throws RemoteException {
    if (this.currentState == State.READ_SHARED) {
        this.currentState = State.INVALID;
        return true;
    }
    return false;

}

// rmi call for the server to request client to release
// the write ownership
```

```
    public boolean writeback() throws RemoteException {
        if (this.currentState == State.WRITE_OWNED) {
            this.currentState = State.RELEASE_OWNERSHIP;
            return true;
        }
        return false;
    }

    // enum State that used to track the state of the cache files
    private enum State {
        INVALID, READ_SHARED, WRITE_OWNED, RELEASE_OWNERSHIP;
    }
}
```

FileServer.java

```
import java.io.*;
import java.net.PortUnreachableException;
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
import java.util.Vector;

/**
* FileServer for css434 final project
*
* @auhtor Haram Kwon, Kris Kwon
* @version 0.1
*/
public class FileServer extends UnicastRemoteObject implements ServerInterface {
   private Vector<File> files = null;
   private int port = 0;


   /**
    * @param port
    * @throws RemoteException
    */
   public FileServer(int port) throws RemoteException {
       this.port = port;
       this.files = new Vector<>();
   }

   /**
    * @param args
    */
   public static void main(String[] args) {
       try {
           if (args.length != 1) {
```

```
            String[] message = new String[1];
            message[0] = "new String[0] = usage: java Server port";
            throw new IllegalArgumentException(
                    "new String[0] = usage: java Server port");
        }

        // Now get the port number
        int port = Integer.parseInt(args[0]);

        if (port < 5001 || port > 65535) {
            throw new PortUnreachableException(
                    "port range should be 5001 ~ 65535");
        }

        startRegistry(port);
        FileServer serverObject = new FileServer(port);
        Naming.rebind(
                "rmi://localhost:" + port + "/fileserver", serverObject);
        System.out.println("Server ready.");

    } catch (PortUnreachableException e) {
        e.printStackTrace();
        System.exit(-1);
    } catch (IllegalArgumentException e) {
        e.printStackTrace();
        System.exit(-1);
    } catch (Exception e) {
        e.printStackTrace();
        System.exit(-1);
    }

}

/**
 * Start the rmi registry
 *
 * @param port the port number for the server
```

```
 * @throws RemoteException rmiregistry start failed
 */
private static void startRegistry(int port) throws RemoteException {
    try {
        Registry registry = LocateRegistry.getRegistry(port);
        registry.list();
    } catch (RemoteException e) {
        Registry registry = LocateRegistry.createRegistry(port);
    }
}

/**
 * Client invoke this function to download the file.
 *
 * @param client the client it is downloading the file
 * @param filename the filename client requested
 * @param mode r/w (read/write)
 * @return the content of the file
 * @throws RemoteException
 */
public FileContents download(String client, String filename, String mode)
        throws RemoteException {

    // todo: filename error checking should be done here

    File file = null;
    byte[] fileContent = null;

    // Scan the cached file list.
    for (File f : files) {
        if (f.filename.equals(filename)) {
            file = f;
        }
    }

    // file not found, and add file to the list
    if (file == null) {
```

```
            file = new File(filename, this.port);
            this.files.add(file);
        }

        // todo:
        return file.download(client, mode);
    }

    /**
     * Client invoke this funtion to upload the file to the server.
     *
     * @param client
     * @param filename
     * @param contents
     * @return
     * @throws RemoteException
     */
    public boolean upload(String client, String filename, FileContents contents)
            throws RemoteException {
        System.out.println("upload invoked");
        File file = null;

        // file the file to upload
        for (File f : files) {
            if (filename.equals(f.filename)) {
                System.out.println(
                        "Trying to find " + filename + " now: " + f.filename);
                file = f;
                break;
            }
        }

        System.out.println("escaped for loop");
        System.out.println("filename: " + file.filename);

        return (file != null) && file.upload(client, contents);
    }
```

```
// States of the files
enum State {
    NOT_SHARED, READ_SHARED, WRITE_SHARED, OWNERSHIP_CHANGE
}

/**
 *
 */
private class File {

    private State state;
    private String filename;
    private byte[] bytes = null;
    private Vector<String> readers = null;
    private String owner = null;
    private int port = 0;

    // Two monitors for handling the clients write access.
    private Object monitor1 = null;
    private Object monitor2 = null;

    /**
     * File constructor for each file.
     *
     * @param filename name of the file.
     * @param port part for rmi registry.
     */
    public File(String filename, int port) {
        this.state = State.NOT_SHARED;
        this.filename = filename;
        readers = new Vector<String>();
        owner = null;
        this.port = port;
        monitor1 = new Object();
        monitor2 = new Object();
```

```
    // read file contents from the local disk
    bytes = readFile();
}

/**
 * Read file from the file strogae and
 * gives back the contents of the file in byte[] form.
 *
 * @return contents of the file.
 */
private byte[] readFile() {
    byte[] bytes = null;
    try {
        FileInputStream file = new FileInputStream(filename);
        bytes = new byte[file.available()];
        file.read(bytes);
        file.close();
    } catch (FileNotFoundException e) {
        e.printStackTrace();
        return null;
    } catch (IOException e) {
        e.printStackTrace();
        return null;
    }
    System.out.println("file read from " + filename + ": " +
            bytes.length + " bytes");
    return bytes;
}

/**
 * write the cached byte[] (or filecontents) to the local storage.
 *
 * @return
 */
private boolean writeFile() {
    try {
        FileOutputStream file = new FileOutputStream(filename);
```

```
            file.write(bytes);
            file.flush();
            file.close();
        } catch (Exception e) {
            e.printStackTrace();
            return false;
        }

        return true;
    }

    /**
     * remove reader if exist
     */
    private void removeReader(String client) {
        readers.remove(client);
    }

    /**
     * Gives the file contents to the client.
     * (helperf function for FileServer.download)
     *
     * @param client the clinet
     * @param mode r/w (read/write)
     * @return
     */
    public FileContents download(String client, String mode) {
        try {
            if (mode.equals("r")) {
                System.out.println("read mode");
            } else if (mode.equals(("w"))) {
                System.out.println("write mode");
            } else {
                System.err.println("mode error with " + mode);
                return null;
            }
```

```
// Thread control for OWNERSHIP change (invoke monitor)
synchronized (monitor1) {
    if (state == State.OWNERSHIP_CHANGE) {
        // todo: delete later
        System.out.println("wait state for ownershiop change");
        monitor1.wait();
        System.out.println("Wait state released");
    }
}

// Save the previous state
State previousState = state;

// do the file action according to file state.
switch (state) {
    case NOT_SHARED:
        System.out.println("download state not shared");
        if (mode.equals("r")) {
            state = State.READ_SHARED;
            readers.add(client);
        } else if (mode.equals("w")) {

            state = State.WRITE_SHARED;
            if (owner != null)
                throw new SyncFailedException(
                                "Critical error. " +
                                        "previous owner " +
                                        "exist in " +
                                        "NOT_SHARED file");
            else
                owner = client;
        }
        break;
    case READ_SHARED:
        System.out.println("download state read shared");
        removeReader(client);
        if (mode.equals("r"))
```

```
            readers.add(client);
        else if (mode.equals("w")) {
            state = State.WRITE_SHARED;
            if (owner != null)
                throw new SyncFailedException(
                        "Critical error. previous owner " +
                                "exist in READ_SHARED file");
            else
                owner = client;
        }

        break;
    case WRITE_SHARED:
        System.out.println("download state write shared");
        removeReader(client);
        if (mode.equals("r"))
            readers.add(client);
        else if (mode.equals("w")) {
            state = State.OWNERSHIP_CHANGE;
            ClientInterface currentOwner =
                    (ClientInterface) Naming.lookup(
                            "rmi://" + owner + ":" +
                                    port + "/fileclient");

            System.out.println(
                    "from " + owner +
                            " write back is requested");

            // requesting write back from the client
            currentOwner.writeback();

            synchronized (monitor2) {
                monitor2.wait();
            }

            // wait around here, and once
            // owner client upload the file,
```

```
                    //change the owner.
                    owner = client;
                }
                break;
        }

        // retrieve file contents from cache
        FileContents contents = new FileContents(bytes);

        if (previousState == State.WRITE_SHARED) {
            synchronized (monitor1) {
                monitor1.notify();
            }
        }

        return contents;
    } catch (Exception e) {
        e.printStackTrace();
        return null;
    }
}

/**
 * give the file content to the client.
 * (helper function for FileServer.download)
 *
 * @param client the client w
 * @param contents
 * @return
 */
public boolean upload(String client, FileContents contents) {
    System.out.println("upload is called");

    try {
        // invalidate all readers' cache
        ClientInterface clientInterface = null;
        for (String reader : readers) {
```

```
            // RMI registration;
            clientInterface = (ClientInterface) Naming.lookup(
                    "rmi://" + reader + ":" + port + "/fileclient");
            if (clientInterface != null) {
                clientInterface.invalidate();
            }
        }

        // clear readers (subscribers)
        readers.removeAllElements();

        State prev_state = state;

        // save file contents
        bytes = contents.get();
        System.out.println("bytes written = " + new String(bytes));

        // state transition
        switch (state) {
            case WRITE_SHARED:
                state = State.NOT_SHARED;
                owner = null;
                writeFile();
                break;
            case OWNERSHIP_CHANGE:
                state = State.WRITE_SHARED;
                owner = client;
                synchronized (monitor2) {
                    monitor2.notify();
                }
                break;
        }


        return true;
    } catch (Exception e) {
```

```
                e.printStackTrace();
                return false;
            }
        }
    }
}
```