Justin Hastings & Jared Rothstein

Spring 2018 Undergraduate Research Write-Up

**Overview:**

The goal of this research was to develop a secure Electronic Voting System that utilized both a client and server interface. In order to do this, we simulated both the client and server using VirtualBox. Each virtual machine was given about 10 GB of hard disk space, 2 GB of RAM, and ran the Ubuntu operating system. In order for the client and server to communicate with another, we also used some tools in VirtualBox to simulate a USB drive that was plugged into both the client and the server. Once we had the basic necessities setup we began to develop code to provide the functionality detailed in the spec provided by Professor Vorobeychik. We choose to code our EVS in Java as it was a language we were both familiar writing scripts in, and it is known to be a safe language. Before coding we decided to made sure to handle certain dependencies and libraries by installing the necessary java tools on both the client and server. Specific details of our implementation of the server, client, and USB drive can be found below.

**Server**

Figure 1.1 to the right shows our main method for the server. We used main to parse user input and determine which commands to run. The -cmd flag notified the server that the user either wanted to configure the server, calculate results, exit the server, or score the teams for this round. The -cgi flag notified the server the user wanted a Common Gateway Interface to run the server. If an improper user input was passed we alerted the user that their command was invalid. As you can see from the else-if for CGI mode we decided to use GUI to simulate CGI rather than HTML as we were

Figure 1.1

```
public static void main(String[] args) {
    // Determine which command is called
    if (args[0].equals("-cmd")) {
        if (args[1].equals("configure")) {
            configure();
        } else if (args[1].equals("results")) {
            results();
        } else if (args[1].equals("exit")) {
            exit();
        } else if (args[1].equals("score")) {
            score();
        } else {
            System.out.println("Command not valid. Try again");
        }
    } // Used to enter CGI simulated mode
    else if (args[0].equals("-cgi")) {
        CGI_MODE = true;
        JFrame frame = new JFrame("serverGUI");
        frame.getContentPane().add(new server().server);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.pack();
        frame.setVisible(true);
    } // Alerts user that an invalid command was input
    else {
        System.out.println("Command not valid. Try again");
    }
}
```

more familiar with how to use java GUI libraries and tools.

Figure 1.2 to the right shows our implementation of the **CGI** simulation using GUI. If CGI mode is selected from the main method, the IF statement is entered and the GUI components are displayed. The dimensions of the GUI components were hardcoded rather than using a GUI form so that the GUI could be properly launched from the command line. Action Listeners were then used to determine which functions to call when the buttons of the GUI were pressed.

```
public server() {
    if(CGI_MODE) {
        server = new JPanel();
        server.setLayout(new GridLayout(1, 0, 10, 0));
        configureButton = new JButton("configure");
        resultsButton = new JButton("results");
        score = new JButton("score");
        exit = new JButton("exit");
        server.add(configureButton);
        server.add(resultsButton);
        server.add(score);
        server.add(exit);


        configureButton.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                configure();
                JOptionPane.showMessageDialog(null, "'configure' has processed", "Message:", JOptionPane.INFORMATION_MESSAGE);
            }
        });


        resultsButton.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                results();
                JOptionPane.showMessageDialog(null, "'results' has processed", "Message:", JOptionPane.INFORMATION_MESSAGE);
            }
        });

        exit.addActionListener(new ActionListener(){
            @Override
            public void actionPerformed(ActionEvent e) {
                exit();
                System.exit(0);
                JOptionPane.showMessageDialog(null, "'exit' has processed", "Message:", JOptionPane.INFORMATION_MESSAGE);
            }
        });

        score.addActionListener(new ActionListener(){
            @Override
            public void actionPerformed(ActionEvent e) {
                score();
                JOptionPane.showMessageDialog(null, "'score' has processed", "Message:", JOptionPane.INFORMATION_MESSAGE);
            }
        });
    }
}
```

Figure 1.3

```
public static void configure() {
    String fileName = "/evs/config/config.txt";
    String line = null;
    try {
        FileReader fileReader = new FileReader(fileName);
        BufferedReader bufferedReader = new BufferedReader(fileReader);
        File fout = new File("/evs/usb/configOut.txt");
        FileOutputStream fos = new FileOutputStream(fout);
        BufferedWriter bw = new BufferedWriter(new OutputStreamWriter(fos));

        // Creates a file that will be passed to the client that contains the configuration information
        while((line = bufferedReader.readLine()) !=null){
            String[] eachAttr = line.split(" ");
            try {
                if (Integer.parseInt(eachAttr[0]) >= 0 && Integer.parseInt(eachAttr[2]) >= 0) {
                    bw.write(line);
                    bw.newLine();
                }
            }catch(NumberFormatException e){

            }
        }
        bw.close();
    }catch(FileNotFoundException e){
        System.out.println("Unable to open file '" + fileName + "'");
    }catch(IOException e){
        e.printStackTrace();
    }
}
```

Figure 1.3 to the left shows our implementation of the servers **configure** function. The server read in the config.txt file placed in the config folder and copied the contents to a file called configOut.txt that was placed on the shared USB drive between the client and the server. This was done so that

the client may have access to the configuration information of this voting round. We used FileReader and BufferReader to read from the config.txt file and then used a FileOutputStream as well as a BufferedWriter to write the configOut.txt file.

Figure 1.4 to the right shows our implementation of the servers **results** function. To start we used a map to tally the results of each candidate and initialized it using the helper function loadPrevTally that initialized the map to the current tally or to zero if there were no previous voting sessions. We then checked to see if a file called prevTally.txt existed. prevTally.txt is a file produced by our server whenever the results function is called which stores the tally from the last time the results function was called. If this file exists a function called checkRepeatedInvocations was called which checked to see if the votes being tallied were identical to the previous invocation of the results function. If it was determined it was a repeated invocation the results function terminated and tally.txt remained the same, however, if this was a new invocation tally.txt would be updated with new votes. As with the configure function we used BufferedReaders and BufferedWriters to read the votes from the usb provided by the client and to write the output tally.txt. At the end of the results function a check is there to see if tally.txt was produced. If it was not produced, the server is alerted there was a DOS attack and produces a file called alert.txt in the /evs/alert directory.

Figure 1.4

```
public static void results() {
    Map<Integer, Integer> talleyMap = loadPrevTalley();
    String talleyVotesFile = "/evs/usb/talleyVotes.txt";
    String line = null;
    try{
        File prevTally = new File("/evs/serverFiles/prevTally.txt");
        // A check to see if there has already been a tally conducted this round, and if so make sure that this is
        // tally does not match the previous.
        if(prevTally.exists()){
            if(checkRepeatedInvocations(talleyVotesFile,prevTally.getPath())) {
                return;
            }
        }
        FileOutputStream fos = new FileOutputStream(prevTally);
        BufferedWriter bwPrevFile = new BufferedWriter(new OutputStreamWriter(fos));

        FileReader fileReader = new FileReader(talleyVotesFile);
        BufferedReader bufferedReader = new BufferedReader(fileReader);

        File fout = new File("/evs/config/tally.txt");

        FileOutputStream fos2 = new FileOutputStream(fout);
        BufferedWriter bw = new BufferedWriter(new OutputStreamWriter(fos2));

        // Keep track of tallies in a map
        while((line = bufferedReader.readLine()) != null){
            try {
                int id = Integer.parseInt(line);
                bwPrevFile.write(line); // should we do this here?
                bwPrevFile.newLine();
                if(talleyMap.containsKey(id)){
                    talleyMap.put(id, talleyMap.get(id)+1);
                }else{
                    talleyMap.put(id,1);
                }
            }catch(NumberFormatException e){

            }
        }
        Iterator it = talleyMap.entrySet().iterator();

        // Write the final tally to the tally.txt file stored in the config folder on the server
        while(it.hasNext()){
            Map.Entry pair = (Map.Entry)it.next();
            bw.write(pair.getKey().toString()+ " " + pair.getValue().toString());
            bw.newLine();
        }
        bw.close();
        bwPrevFile.close();
        // Alert the server that there is a DOS attack and tally.txt was not produced
        if (!fout.exists()) {
            File alert = new File ("/evs/alert/alert.txt");
            FileOutputStream sAlert = new FileOutputStream(alert);
            BufferedWriter alertBW = new BufferedWriter(new OutputStreamWriter(sAlert));
            alertBW.write("dos\n");
            alertBW.close();
        }
```

```
public static void exit() {

    File tally = new File ("/evs/config/tally.txt");
    File stally = new File ("/evs/scoring/tally.txt");
    File alert = new File ("/evs/alert/alert.txt");
    File score = new File ("/evs/scoring/score.txt");
    File prevTally = new File("/evs/serverFiles/prevTally.txt");
    tally.delete();
    stally.delete();
    alert.delete();
    score.delete();
    prevTally.delete();

}
```

Figure 1.5 to the left shows the **exit** function that was used to delete files produced by the server after a round was complete. This very simply called the delete function from the File class on all appropriate files so that the next round would have a clean slate.

Figure 1.6 to the right shows our implementation of the **score** function. In order to score if a team was able to successfully attack the EVS we have the white team place the correct tally.txt in the /evs/scoring directory. From here the score function uses the checkRepeatedInvocation function, which basically identifies if two files are identical, to determine if there was an attack. If the proper tally.txt file was produced the EVS team received 2 points and if not the red team received a single point. This function also handle special scoring for DOS attacks whether they go undetected, detected, or there is a false alarm. If there is a false alarm or a DOS attack goes undetected the red team receives a point and if a DOS attack is properly detected the DOS team receives a point. The results of the scoring are placed in a text file called score.txt in the /evs/scoring directory.

```
public static void score() {
    int bteam = 0;
    int rteam = 0;
    int dosAlert = 0;
    int dosNoAlert = 0;
    int dosFalseAlert = 0;

    try {
        File fout = new File("/evs/scoring/score.txt");
        FileOutputStream fos = new FileOutputStream(fout);
        BufferedWriter bw = new BufferedWriter(new OutputStreamWriter(fos));
        String tally = "/evs/config/tally.txt";
        File tallyf = new File (tally);
        String ctally = "/evs/scoring/tally.txt";
        File alert = new File ("/evs/alert/alert.txt");
        if (tallyf.exists()) {
            // Checks if tally.txt is correct for the round. If it is the EVS team receives two points, if it is not
            // the red team receives one point.
            if (checkRepeatedInvocations(tally,ctally)) {
                bteam += 2;
                // If alert.txt is created and tally.txt was actually correct the red team receives a point for
                // triggering a false alarm
                if (alert.exists()) {
                    dosFalseAlert++;
                }
            } else {
                rteam++;
                // If tally.txt is incorrect and no DOS alert is triggered then the red team receives a point for
                // performing a DOS attack undetected
                if (!alert.exists()) {
                    dosNoAlert++;
                }
            }

        } else {
            // The EVS team receives a point for detecting that there was a DOS attack when appropriate
            if (alert.exists()) {
                dosAlert++;
            }
        }
        bw.write("T\t\tB\tR\n");
        bw.write("1\t\t" + bteam + "\t0\n");
        bw.write("0\t\t0\t" + rteam + "\n");
        bw.write("DOS(NA)\t0\t" + dosNoAlert + "\n");
        bw.write("DOS(A)\t" + dosAlert + "\t0\n");
        bw.write("DOS(FA)\t0\t" + dosFalseAlert + "\n");
        bw.close();
    } catch (FileNotFoundException e) {
        System.out.println("Unable to open file\n");
    } catch(IOException e){
        e.printStackTrace();
    }
}
```

```
public static boolean checkRepeatedInvocations(String votes,String preVotes){
    String line1 = null, line2 = null;
    try {
        FileReader votesReader = new FileReader(votes);
        FileReader prevVotesfileReader = new FileReader(preVotes);
        BufferedReader votesbufferedReader = new BufferedReader(votesReader);
        BufferedReader prevbufferedReader = new BufferedReader(prevVotesfileReader);
        line1 = votesbufferedReader.readLine();
        line2 = prevbufferedReader.readLine();
        while(line1!=null && line2!=null && line1.equals(line2)){
            line1 = votesbufferedReader.readLine();
            line2 = prevbufferedReader.readLine();
        }
        if(line1==null && line2==null)
            return true;
    }catch (FileNotFoundException e){

    }
    catch (IOException e){

    }
    return false;
}

public static Map<Integer,Integer> loadPrevTalley(){
    Map<Integer, Integer> prevTalleyMap = new HashMap<>();
    String line = null;
    try {
        String prevtalleyVotesFile = "/evs/serverFiles/prevTally.txt";
        File file = new File(prevtalleyVotesFile);
        if (!file.exists()) {
            return prevTalleyMap;
        }
        FileReader fileReader = new FileReader(prevtalleyVotesFile);
        BufferedReader bufferedReader = new BufferedReader(fileReader);
        while((line = bufferedReader.readLine()) != null){
            try {
                int id = Integer.parseInt(line);
                if(prevTalleyMap.containsKey(id)){
                    prevTalleyMap.put(id, prevTalleyMap.get(id)+1);
                }else{
                    prevTalleyMap.put(id,1);
                }
            }catch(NumberFormatException e){

            }
        }

    }catch(FileNotFoundException e){

    }catch (IOException e){

    }
    return prevTalleyMap;

}
```

Figure 1.7

Figure 1.7 to the left shows the helper functions that were used in the configure and score functions. checkRepeatedInvocations takes in two file paths as strings and returns whether or not the files are identical. This is used in configure to determine if the same votes were passed to the server to be tallied again, as well as in score to determine if the tally.txt that the server produced was correct according to the tally.txt provided by the white team. The second function loadPrevTally was used to initialize the map every time results was called either to the current tally or a empty map if there were no previous voting sessions.

**Client**

```
public static void main(String[] args) {
    if (args[0].equals("-cmd")) {
        if (args[1].equals("load")) {
            load();
        } else if (args[1].equals("run")) {
            run();
        } else if (args[1].equals("exit")) {
            exit();
        } else {
            System.out.println("Command not valid. Try again");
        }
    } else if (args[0].equals("-cgi")) {
        CGI_MODE = true;
        JFrame frame = new JFrame("clientGUI");
        frame.getContentPane().add(new client().clientGUIPanel);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.pack();
        frame.setVisible(true);
    } else {
        System.out.println("Command not valid. Try again");
    }

}
```

Figure 2.1

Figure 2.1 to the left shows our main method for the client. We used the main method to parse user input and determine which commands to run. The -cmd flag notified the client that the user either wanted to load the configuration information for the current election from the USB drive, wanted to read votes.txt from the client's /votes directory by executing 'run', or wanted to exit the program. The -cgi flag notified the client the user wanted a Common Gateway Interface to run the client. If an improper user input was passed we alerted the user that their command was invalid.

```
public client() {
    if(CGI_MODE) {
        clientGUIPanel = new JPanel();
        clientGUIPanel.setLayout(new GridLayout(1, 0, 10, 0));
        loadButton = new JButton("load");
        runButton = new JButton("run");
        exit = new JButton("exit");
        clientGUIPanel.add(loadButton);
        clientGUIPanel.add(runButton);
        clientGUIPanel.add(exit);

        loadButton.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                load();
                JOptionPane.showMessageDialog(null, "'load' has processed", "Message:", JOptionPane.INFORMATION_MESSAGE);
            }
        });

        runButton.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                run();
                JOptionPane.showMessageDialog(null, "'run' has processed", "Message:", JOptionPane.INFORMATION_MESSAGE);
            }
        });

        exit.addActionListener(new ActionListener(){
            @Override
            public void actionPerformed(ActionEvent e) {
                exit();
                System.exit(0);
                JOptionPane.showMessageDialog(null, "'exit' has processed", "Message:", JOptionPane.INFORMATION_MESSAGE);
            }
        });
    }
}
```

Figure 2.2

Figure 2.2 on the right shows our implementation of the CGI interface for the client. If CGI mode is selected from the main method, the IF statement is entered and the GUI components are displayed. In order to be able to launch the GUI from the command line, all GUI components had to be hardcoded. The three ActionListeners are used to implement the functionality of each button when they are clicked. Each ActionListener displays a message box notifying the user that the function selected has finished processing.

```
public static void load() {
    String fileName = "/evs/usb/configOut.txt";
    String line = null;
    try {
        FileReader fileReader = new FileReader(fileName);
        BufferedReader bufferedReader = new BufferedReader(fileReader);
        File fout = new File("/evs/clientFiles/configInfo.txt");
        FileOutputStream fos = new FileOutputStream(fout);
        BufferedWriter bw = new BufferedWriter(new OutputStreamWriter(fos));
        while((line = bufferedReader.readLine()) != null){
            String[] params = line.split(" ");
            try {
                int id = Integer.parseInt(params[0]);
                int votes = Integer.parseInt(params[2]);
                if (id >= 0 && votes>=0) {
                    bw.write(id + " " + votes);
                    bw.newLine();
                }
            }catch(NumberFormatException e){

            }
        }
        bw.close();
    }catch(FileNotFoundException e){
        System.out.println("Unable to open file '" + fileName + "'");
    }catch(IOException e){
        e.printStackTrace();
    }
}
```

Figure 2.3

Figure 2.3 to the left shows the load function of the client. This was used to load the configuration information passed to the client from the server via the virtual usb drive. As seen before, we used BufferedReaders and BufferedWriters to read in the config.txt file and create a local configInfo.txt file on the client that contained the necessary configuration information for the client to run properly.

Figure 2.4 to the right shows our implementation of the run function for the client. A helper function called initializeConfigInfo, as described below, is first called to initialize the configuration information into a HashMap for easy access. The first if statement checks to see if a previous votes file exists, and if it does, it called the checkRepeatedInvocations method to determine if the run command was invoked multiple times with the same sequence of votes. If the sequence of votes has stayed the same, this IF statements escapes out of the method. The while loop is then used to read the votes.txt file and output the results of the voting session. The hashmap is used to determine whether a candidate ID is valid in this process. This method then outputs a file to the USB with the results.

```
public static void run() {
    initializeConfigInfo();
    String votes = "/evs/votes/votes.txt";
    String line = null;
    try {

        File prevVotes = new File("/evs/clientFiles/prevVotes.txt");
        if(prevVotes.exists()){
            if(checkRepeatedInvocations(votes,prevVotes.getPath())) {
                return;
            }
        }
        FileOutputStream fos = new FileOutputStream(prevVotes);
        BufferedWriter bwPrevFile = new BufferedWriter(new OutputStreamWriter(fos));

        File talleyVotes = new File("/evs/usb/talleyVotes.txt");
        if(!talleyVotes.exists()){
            talleyVotes.createNewFile();
        }
        FileWriter talleyfos = new FileWriter(talleyVotes,true);
        BufferedWriter talleybw = new BufferedWriter(talleyfos);
        FileReader votesfileReader = new FileReader(votes);
        BufferedReader bufferedReader = new BufferedReader(votesfileReader);
        while((line = bufferedReader.readLine()) != null){
            try {
                int id = Integer.parseInt(line);
                if (id>=0 && configInfo.containsKey(id)) {
                    bwPrevFile.write(id+"");
                    bwPrevFile.newLine();
                    talleybw.write(id+"");
                    talleybw.newLine();
                }
            }catch(NumberFormatException e){

            }
        }
        bwPrevFile.close();
        talleybw.close();
    }catch(FileNotFoundException e){
        System.out.println("Unable to open file '" + votes + "'");
    }catch(IOException e){
        e.printStackTrace();
    }
}
```

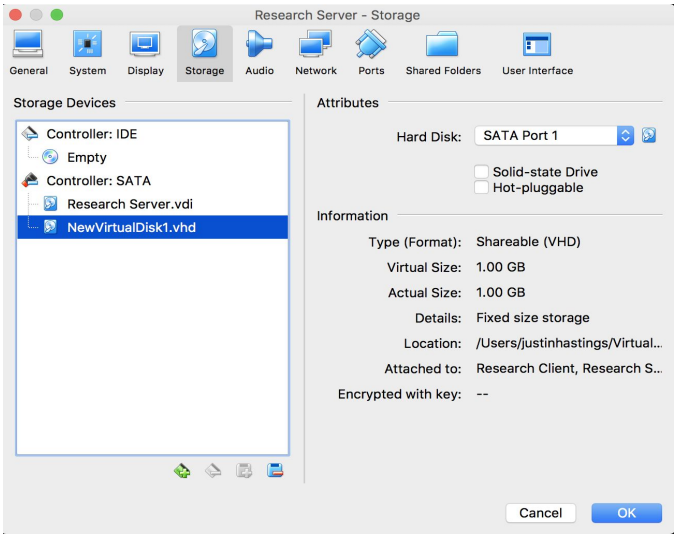
```java
public static void initializeConfigInfo(){
    configInfo = new HashMap<>();
    String fileName = "evs/clientFiles/configInfo.txt";
    String line = null;
    try {
        FileReader fileReader = new FileReader(fileName);
        BufferedReader bufferedReader = new BufferedReader(fileReader);
        while((line = bufferedReader.readLine()) != null) {
            String[] params = line.split(" ");
            try {
                int id = Integer.parseInt(params[0]);
                int votes = Integer.parseInt(params[1]);
                configInfo.put(id,votes);
            }catch(NumberFormatException e){

            }
        }
    }catch(FileNotFoundException e){
        System.out.println("Unable to open file '" + fileName + "', load command was not executed prior to run command");
    }catch(IOException e){
        e.printStackTrace();
    }
}

public static boolean checkRepeatedInvocations(String votes,String preVotes){
    String line1 = null, line2 = null;
    try {
        FileReader votesReader = new FileReader(votes);
        FileReader prevVotesfileReader = new FileReader(preVotes);
        BufferedReader votesbufferedReader = new BufferedReader(votesReader);
        BufferedReader prevbufferedReader = new BufferedReader(prevVotesfileReader);
        line1 = votesbufferedReader.readLine();
        line2 = prevbufferedReader.readLine();
        while(line1!=null && line2!=null && line1.equals(line2)){
            line1 = votesbufferedReader.readLine();
            line2 = prevbufferedReader.readLine();
        }
        if(line1==null && line2==null)
            return true;
    }catch (FileNotFoundException e){

    }
    catch (IOException e){

    }
    return false;
}
```

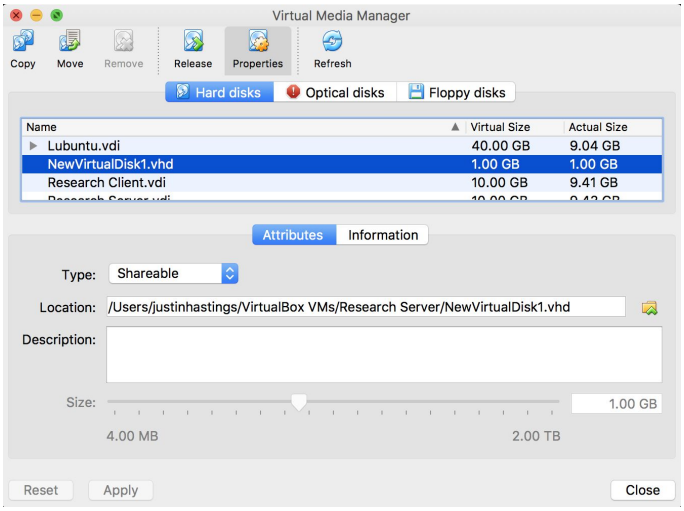
Figure 2.5

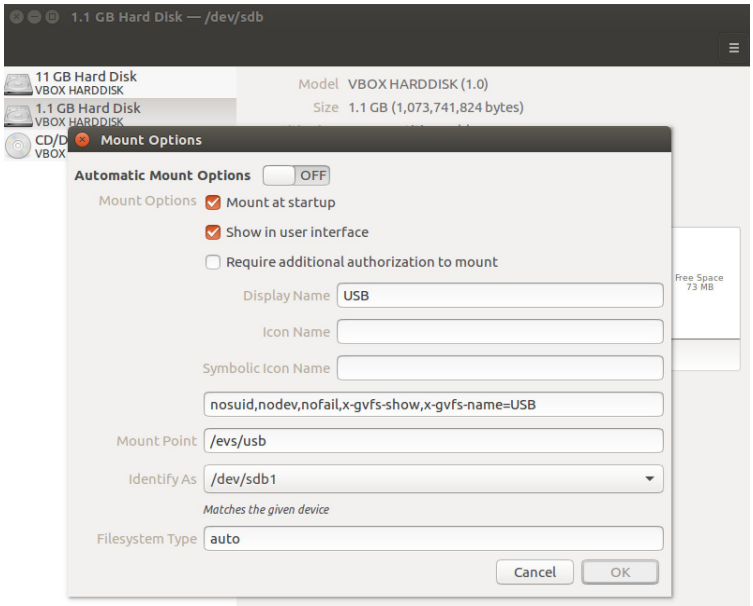
Figure 2.5 on the left shows two helper methods initializeConfigInfo() and checkRepeatedInvocations(). initializeConfigInfo() is used to store the configuration information into a hashmap used for error checking. This method stores all candidate ID's and their initial votes into the hashmap which is then later used to determine if candidate IDs are valid. The checkRepeatedInvocations method is used to determine if the run command was invoked multiple times with the same sequence of votes. This method takes in two parameters, the path of the current voting session and the path of the previous voting session. The while loop then goes line by line between the current and previous voting session and determines if the equivalent. This method returns a boolean answer, true if the files are the same, false if the are not.

## USB

In order to create a simulation of a USB drive that is shared between the server and client for communication we used the VirtualBox storage tools. The figure to the right shows the fixed size storage drive that was created in VirtualBox and given to the server. First we opened the servers setting in VirtualBox and navigated to the storage pane. We then made the disk only 1 GB as the USB should only ever hold two text files. After creating this virtual drive and designating it to the server we then closed out this pane. We then navigated to “File | Virtual Media Manager” to open up the pane shown on the left. Within this pane we were then able to set the newly created USB drive to be “Shareable” so that it could later be accessed by two virtual machines concurrently. Once this was complete we then added this drive to the client in a similar fashion as it was originally added to the server. We then used the Ubuntu Disk Manager to set the settings to mount this USB drive in both the client and the server to the /evs/usb directory upon startup. These setting can be seen in the screenshot below.



**Testing**

To test our implementation of the EVS you must have VirtualBox installed. You can then open the JJResearch.ova file in the github repository and import both the client and server to your VirtualBox. This should come loaded with the necessary files on each virtual machine as well the shared usb drive. If for some reason the usb device does not load properly, you can follow the above steps to reconfigure this. Once you have both virtual machines properly installed you can deploy both machine simultaneously from VirtualBox. From there, navigate to the /evs/server and /evs/client directories in the appropriate machines from the terminal. In the terminal run the commands “javac <server.java/client.java>” to build the programs. Then to run any commands such as configure, load, run, results, score, and exit simply type “java <server/client> <-cmd/-cgi> <configure/load/run/results/score/exit>”