Opportunistic Multimedia Dropbox
Multibox

TEAM 03

Caprini Carlo (150124), Zeni Mattia (150082)

Master Degree in Telecommunication Engineering, Faculty of Trento, Italy

June 21, 2012

Abstract

This project regards the realization of a *Dropbox- like application* for multimedia content. We named it *Multibox* and we developed it for Mac OS X 10.7 platforms.

Thanks to this application, users can synchronize and share any contents saved in pre-defined directories keeping each file always up to date to the most recent version. This is possible because special functions have been created in order to confront the information of each file with the purpose of verify the presence or absence of the multimedia contents and to check their version.

A key feature of this application is its capability of exploiting opportunistically transmission opportunities. This means the program is able to start synchronizing multimedia contents with another device as soon as a connection with a network has been established and a second user running the application and belonging to the friend's list has been found.

Once it has been launched, the application is autonomous and the user does not have to start, stop or pause the synchronization process, no matter the presence or absence of networks or friends.

1. Objectives

The aim of this application is to enable users to always keep the files they are sharing up to date. This is possible because, during every synchronization phase, all the information about the files saved into the shared directory, are sent to the connected user, which is going to process them in order to define if the file is missing, if it is an older or a newer version, if the file has been renamed or if it simply is salready up to date. All this possibilities can be distinguished thanks to the three parameters *name of the file*, *hash of the file* and *last modification date of the file*.

A second key feature of the application is its capability of being able to synchronize the files in any place the user can be found together with the ability of being totally autonomous. In fact, once it is launched, the user can continue to work without giving it a second thought. Also in the case in which the computer enters sleep mode and later reactivates or connect to a different network the application does not need any interaction with the user. As soon as there is the chance, Multibox automatically tries to contact as many users as it is possible and starts synching right away.

When launched, if no known network connections can be found, the application creates its own ad-hoc one and waits there for new incoming requests. Therefore there are no places where two users can not synchronize their files with extreme simplicity.

Once correctly connected to a network, the application starts looking for other users connected to the same network by sending a certain number of UDP *broadcast hello messages*. If no answers are received, the application enters a *listening mode* where it waits for UDP *broadcast hello messages* coming from newly connected users. Else, if an answer is received or the application in *listening mode* receives a *hello message*, the application starts three brand new processes that will be needed for the synchronization phases and for constantly checking the state of the connection. UDP broadcast is needed because the application looking for other users does not know in advance their IP address on the network. Once the first contact has been established, the unreliable UDP connection is abandoned and a reliable TCP one is open between sockets.

It is important to point out that, once the application has been started, the computer is not going to start a synchronization process with any other machines running the same application. During the search phase everyone is obviously going to be contacted but only the users with which a sharing pact has been established are going to start synching with the application.

2. Main Achivements

We have been able to successfully establish a connection and start different synchronization processes when multiple users are connected to the same network. This has been possible because each synchronization process takes place on its own TCP connection, each characterized by a different socket port number. Files belonging to different shared directories can therefore be checked, compared and transferred at the same time.

A correct synchronization is guaranteed also in the nowadays unlikely possibility in which the two computers connected are not set to the same date or to the same hour. In fact, as soon as any two computers contact each other, they exchange their current reference date calculated in seconds from January 1st 1970 00:00 and compute the delay between the two. When comparing the information about the files this delay is always taken into account.

As already mentioned earlier, Multibox supports the synchronization over both infrastructure and ad-hoc networks. When launched, the application firstly checks if the default ad-hoc network has already been created by some other user and if this is the case it connects directly to it. Else, Multibox looks for favorite networks and, if there are none available, it creates the default ad-hoc network and enters in listening mode.

Users can be easily distinguished by looking at their identification code, which is a number of 10 digits generated randomly as soon as the Multibox account is created. In this way, user can choose the same user name while remaining univocally recognizable by looking at their identification code. Through the GUI is very easy to manage the friends (it is possible to both add and remove them) with whom share a directory and only the users which have been added to the friend's list will be able to synchronize the contents.

HTTP transfer has been chosen, therefore the directories that can be shared should be placed into the directory Sites in the home directory. HTTP is very fast and it is very easy for the user to enable the HTTP server.

The last important feature worth mentioning is the ability of the application to resume the previously interrupted download of a file without the need of starting it all over again. Multimedia files can, in fact, reach significant dimensions (just think about an HD video), the time needed to transfer them can be high and it is not unusual that the transfer can be interrupted. We managed to discover the presence of incomplete files in the shared directories and our code is capable of restarting the transfer of the file from where it was interrupted the last time the two users involved where synchronizing.

Unfortunately, Multibox shares one of well-known weaknesses of Dropbox, the application from which it is taking inspiration. Errors in the synchronization are expected in the case in which two users, that are actively running a synchronization process, try to modify at the same time the same file or if one of them tries to delete a file, which is currently open on the computer of the other user. If such situations take place, the synchronization of the target files can unfortunately be lost. This application could become very handy in situations that involve a group work, in which every person is supposedly working on a different file, therefore this unwanted problem can be easily avoided.

3. Activities Performed

The project has been scheduled into the six following activities:

a. **Analysis of the Objective**

Objective: During this first step, the different possible approaches have been studied and the platform and the programming languages have been chosen.

Achievements and results: The first choice regarded the backbone of the application, that is the strategy needed in order to successfully share multimedia content. A possible solution could have been to directly transfer the contents between two devices connected to the same wireless network. The transmission would have been very fast but the users would have needed to be connected to the same network (this means they would have needed to be close to each other). The alternative would have been to use an intermediate server to and from which upload and download the contents. Our final choice has been the first option. Multibox can therefore be used when connected to a network without Internet connection and it can become very handy when working to a group-project, that usually finds members working in the same space. The connection can be established in both ad-hoc or infrastructure modality.

Mac OS X has been chosen as designed platform and C and Objective-C have been chosen as programming languages for the main body of the application and for the GUI respectively.

b. **Study of the Communication Approaches**

Objective: Choose the best synchronization and transfer techniques.

Achievements and results: Devices connected to the same network can be aware of each other presence thanks to the transmission of UDP *broadcast hello messages*.

In order to correctly synchronize the information about the contents saved on the two devices connected, messages are exchanged on a secure TCP connection created between sockets. It is possible to compare files thanks to their names, their last modification date and their hash code.

As the designed platform is Mac OS X, it is very easy to set up a HTTP server. So, files are shared by means of HTTP transmissions. HTTP has been preferred to FTP because it does not need authentication before starting to transfer and because it needs the user to set the preferencies.

c. **Development of the synchronization stage**

Objective: Implementation of the code needed for the synchronization of the devices.

Achievements and results: The outcome of this activity is represented by all the functions needed to the application in order to decide which files are to be transferred between the users. The information about each file saved into the shared directory is sequentially sent

to the other user who is able to understand if a certain file is missing, if its version is up to date, older or newer or if the file has been renamed since last synchronization.

d. **Development of the transfer stage**

Objective: Implementation of the code needed for transferring the files between the devices.

Achievements and results: The result of this activity has been the creation of a simple function that the application calls every time it needs to download a file from the device it is synchronizing with.

For this purpose the free client-side URL transfer library libcurl is used. An important feature of the application resides in the possibility to resume a transfer after a disconnection.

e. **Creation of the User Interface**

Objective: Creation of the GUI.

Achievements and results: A user-friendly user interface has been created. It is very easy and intuitive to create an account to use during the synchronization process, add and remove friends and manage the directories that are to be shared. It is also very easy to know if the application is currently synching with the selected friend or not. Finally it is immediate to know if some files have been modified because Multibox has been integrated with the Growl API and can therefore notify every relevant news.

f. **Test**

Objective: Test the application.

Achievements and results: The application has been tested in order to verify that in all the possible situations it is able to work correctly. Both the transfer and the synchronization stages have been deeply tested in order to perfect the performances of the application.

Thanks to these tests we improved greatly the piece of code needed for calculating the hash of the files, which is a process that can take quite some time. Earlier we used to calculate the hash of every file at the beginning and at the end of every synchronization phase. Now the application calculate the hash only of newly downloaded files while getting the hash of the other files from the *databases* used as *service files*.

Other small errors and missing controls were found but nothing that could not be taken care of.

g. **Final report**

Objectives: Creation of this very Report.

4. Description of Artifax

The application has been developed on two MacBooks running Mac OS X Lion 10.7.3. The entire code has been written in XCode, which can be downloaded directly from the Apple website or through the Mac App Store for free.

The provided software package contains all the source codes.

The main application is composed by multiple files that are here listed and briefly described. main.h contains the definitions of all the functions and all the structures needed by the application. main.c contains one of the two the main parts of the application; it manages the connections with the other users and implements the code run when the program is contacted by a newly connected user. functions.c implements functions with various purposes. transfer.c implements the function needed for transferring the files between the users. Finally md5function.c implements the functions needed for computing the hash of the file and saving it into the *temporary* and *local databases*. In order to create the executable file it is sufficient to launch the provided makefile.

The GUI can be accessed by opening the XCode project Multibox.xcodeproj. The application can in this way be both simulated by running the project or built by following the steps Product->Archive->Distribute->Export As->Application.

The software package contains a .dmg file as well. This is the file the final user should get in order to run the application and contains the application Multibox that should be moved into the Application folder and the folder Multibox which contains all the *service files* needed by the application and that should be moved into the Documents folder. In addition a ReadMe file with the requirements, instructions for the installation and a small guide is present along with a .zip package containing the the libcurl library files. Libcurl should be already be present on the computer; if not, it is sufficient to follow the installation instructions.

When it is launched, the application window appears as in Figure 4.1. It is possible to select

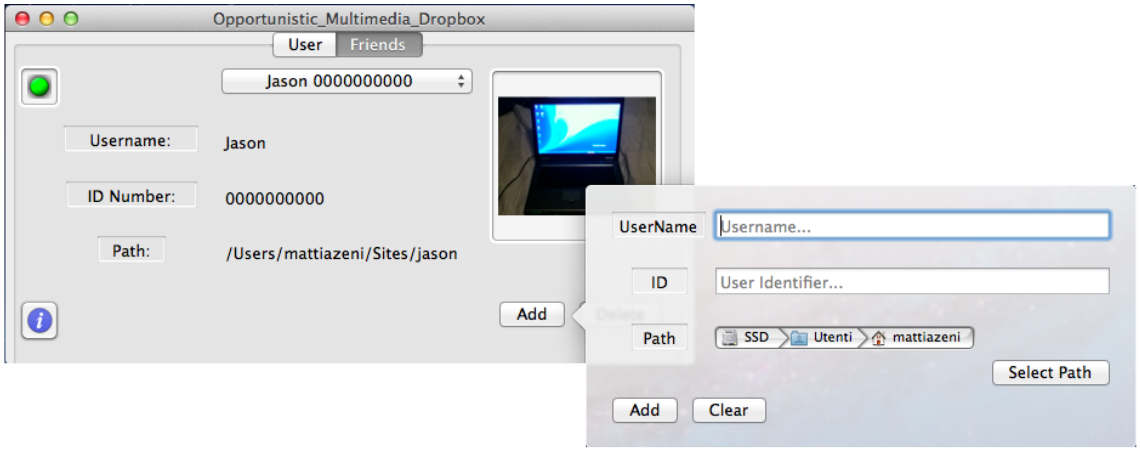

Figure 4.1: Multibox User Interface.

between two different labels. In the User panel it is possible to manage the user information, change the avatar of the user and start or stop the synchronization process by clicking on the two buttons Start and Stop. In the Friends panel it is possible to view the information about all the friends in the friend list such as username, identification code, path of the shared folder and state of the connection displayed by the led on the upper left corner of the window. Thanks to the buttons Add and Delete it is possible to manage the friends adding or removing them. When adding a friend, it is necessary to know not only his user name but also his identification code.

The icon of the application does not appear in the Dock, but, even after the main window has been closed, the more important information can be access by clicking on the Multibox icon in the top menu bar. It is possible to open the main window again, to see the users we are currently synching with, check the available disk space and finally it is possible to start, stop or quit the application.

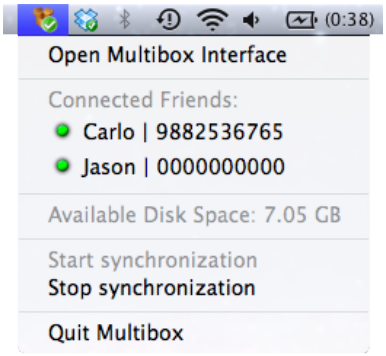

Figure 4.2: Topmenu.

Multibox is also integrated with notification API of Growl. Notifications pop up when the application is started or stopped, but more important they pop up every time a file in one of the shared folders has been updated, downloaded, deleted or renamed. This is a key freature of the application which grants the possibility of not having to worry about the underlying synchronization process. In fact, every time it happens something the user should be informed of, the growl notifications notify the event in a simple and non-invasive way.

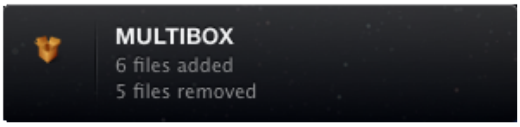

Figure 4.3: Popup Message.

As it has already been mentioned, if no favorite Wlan networks are present when the application is started, Multibox will create its own ad-hoc network to which all the other users will connect at their start. Also in this case the user does not have to perform any operations; it is the application that automatically behaves accordingly to the conditions found in the working environment. When in ad-hoc modality, the file transfer speed is faster than when transferring on an infrastructure network. After some tests we can say that files can be transferred up to 10x faster.

5. Conclusions

In this final chapter, a few ideas that cold be implemented in order to perfect the Multibox application are discussed.

- An important feature we have not been able to develop is the ability to share subfolders. The problem did not reside on the very identification and transfer but on the erase of those subdirectories. One of the subdirectories could then be used to store the avatar of the friends which now appear in the shared directory together with the multimedia content.
- Multibox could be the extension to an ”internet dimension”. In this way, the synchronization could take place also between users places in different locations. It could be interesting to dynamically decide if synching the files over the local network or by using the intermediate server online. A different approach should be studied in order to let the users finding each other also when connected to different networks. A centralized server approach could be used in order to log the connection and disconnection to the service of the different users. Also the system based on broadcast UDP messages that enables the user to find each other on the local networks could be revised. In fact, there could be some networks (as the network of the University of Trento is) that blocks these types of messages.
- With the introduction of the ”on-line dimension” feature, also a web interface could be interesting, in order to better manage the original settings of the application and maybe also new features.
- iOS/Android integration could be also interesting. It could become handy not only for allowing the user to check the status of the synchronization on his own computer but also for sharing the folders on the mobile phone with the ones on the computers.

Annex

In this paragraph the details about the code are going to be highlighted. The GUI will be treated independently from the main application, as the two are two separated entities that can communicate with each other by writing and reading on special files that are saved into Multibox folder, present in the installation package Multibox.dmg.

The code written for the GUI can be divided into three categories: the code for the initialization, the methods which are called every time the user interact with the interface, and the methods which are periodically called by timers.

We need to initialize some parameters such as labels and menus, and we have to check the internet connection. The topmenu has to be created when the application is started and it has to be populated with menu items such as the six static labels (Open, Friend, Disk Space, Start/Stop, Quit). Also the pulldown menu in Friend panel has to be filled with friends usernames and ids. To do that, the application reads *userFriends.txt*, saves the combinations of username and id into a service file *userFriendsNameApp.txt*, and then it reads them and fill the menu. The two labels Username and ID in User panel are filled with user informations that are read from the file *userData.txt*. The third label is the IP one, which is obtained in this initialization process, and the value is put in the field; if no IP is found, it means that no connections are available, and "Disconnected from the Network" is displayed. The IP is also saved into a service file called *userIP.txt* that is used in a timer method, in order to understand if the IP changed, and if that is the case, we restart the synchronization service.

As already mentioned, the application works both on infrastructured and adhoc wifi connection. If no network is present, the first User started creates the adhoc "Multimedia" connection, and the following users connects to it. In order to do that, when the application is initialized, if no IP is found (no network are present) we try to connect to this "Multimedia" adhoc network, and a notification message is sent.

After this activities, we start the three main timers that works periodically every 1, 7 and 15 seconds. The last thing to do in the initialization process is initialize the Drag & Drop items in their respective classes, using the function *registerForDraggedTypes*. One of this, is used to enable drag and drop for user avatar, that copies the dragged image (and only images) into Multibox folder, renaming it to *avatar.jpg/png*, and copies the image also in every shared folder, taking their path from *userFriends.txt* file, renaming them to *avatar'ID'.jpg/png*, where ID is the identification number of every user present in friend service file. The other drag and drop item is used to move shared files released on the dedicated area into the shared folder of the selected friend in the pulldown menu; folder path is obtained from service file *userFriends.txt*.

Every time the user makes an action on the GUI, for example presses a button, or opens a menu, the relative method is called and performs some activities.

Starting from User panel, some actions can be done. We can Start/Stop the synchronization service (also from the top menu we can do it). This methods are mutually exclusive, in the sense that only one button at a time can be selected by the user. In particular, if the service is running and Stop is pressed, you can not press Start again before 15 second because the application needs to delete some files and kill all processes. So if Start is pressed, Stop button is excluded, service file *userPid.txt* is created empty, the synchronization service is called and a notification is displayed. In *userPid.txt* we write all the processes PID numbers, in order to kill them when Stop is pushed. So if you press Stop, this file is read line by line and every process with that PID is killed, a notification is displayed and at the end, some service files are removed, such as *userPid.txt*, *currentFriendsConn.txt*, *currentIpConn.txt*. On User panel we can update our information using the button 'Manage User'. In this case a popup window is opened and the user have just to choose the new Username, because the ID is choosen randomly. The new information are written inside service file *userData.txt*. To update also the avatar, just drag and drop an image on the actual shown avatar. The last thing a user can do in this panel is to see an help file which basically explains briefly the main purpose of each button.

Speaking of Friends panel, the first thing a user have to do is select a friend from the pull-down menu. This menu has been already filled with values in the initialization phase. When you choose a friend from the list, the method is called. It reads service file *userFriends.txt* searching for a corresponding combination of Username-ID selected in the list and if it matches, it fills all the board information and shows the avatar of the user. Notice that Delete button is enabled only if a friend is selected in the menu, and if you push it, a comparison is made with every line in *userFriends.txt* and if the user is present, it removes this line. You can also add a friend pushing on Add button. The relative method is called and popups a window. In this window three fields must be filled: the friend Username, the friend ID and the path of the shared folder (if not present you can create a new one). For the path selection easy system window will popup. Pay attention that the shared folder must be putted inside 'Sites' folder in user directory in order to let the transfer to work. In reality it is not really mandatory, you can simply change the shared folder of Apache HTTP server installed on your Mac from the .config file, but we prefer to not change it because an user can have active websites running, and changing the directory, all websites will not work anymore. The addFriend method, reads the inserted values from the user, compare it with every line of service file *userFriends.txt*, and if there is no match adds the line. When a friend is added, the synchronization service is restarted, in order to let the new friend to immediately share files. Also in Friend there is an Help section which briefly explains the main purpose of each button.

The last category of methods is the one called periodically by timers. We have 7 of them and 3 are called during the initialization phase. These 3 have a timing of 1, 7 and 15 seconds. Basically we decide to use these types of methods in order to update information that need to be constantly up to date and to do controls.

The one second timing method is called dostuff. About the information part, it controls the available disk space and prints it in the item of the topmenu. Then it updates user information panel such as IP, ID, Username and avatar. About the control part, it checks if it is the first time we call the method (if the application have been just started up), if we are connected to a network and if service file *userData.txt* is present (user have filled User information), it starts the synchronization service and popups a message.

The seven seconds timing method is called dostuff7. In here, we update 'dynamic' information in topmenu like connected friends. This method reads service file *currentFriendsConn.txt*

in which we insert actual connected friends line by line. For each line it extracts Username | ID and adds an item in the topmenu with a green check icon on the left of the string. We use this timer also to notificate synchronization status: it periodically checks for existence of service files *added.txt/removed.txt/renemad.txt/updated.txt*, if they exists, it opens them and counts the starts in each file that means that a file has been added, removed, renamed or updated by the friend, and pop up a message.

The fifteen seconds timing method is called dostuff15. This method implements only control features on the network. If the actual IP is different from the previous one, it means that the connection changed, so we need to restart the synchronization service. Also if it is the first time the method is called and no IP is assigned, it means that no networks are present and 'Multimedia' adhoc network has to be created calling an Applescript. If service file error.txt is present, it means that an error occurs in the synchronization service and has to be restarted.

Another timer is the method restartAddFriend called in addFriend, which restarts the synchronization service after 7 seconds. We create this method in order to let the software to completely stop the service, before restarting it.

The complementary timer to add method, is the method restartDeleteFriend called in deleteFriend, which restarts the synchronization service after 7 seconds.

stopEnable timer just enables/disables some buttons.

The last timer used is userStatus method. This method checks *currentFriendsConn.txt* for connected friends, than compares each lines with the actual selected Friend from the pulldown menu and if there is a match, it turns on the green led, otherwise it remains red.

The code written for the main application it is divided among multiple files as it has been explained in the paragraph Description of Artifacts. Every time the interface launches the main application it passes also the home directory path as argv argument. In this way the application now knows how to access the various directories on the machine.

It is very important to immediately point out that every time a new process is created, forking the parent process, its process ID is immediately added to a special file *userPid.txt* so it is always possible to monitor the state of the application. Similarly, every time a process is killed, either because an error occurred or its presence is no longer needed, its process ID is erased from the file.

The first thing that the application does is to initialize all the variables and semaphores needed later. Semaphores are very important because they prevent the possibility that parallel processes managing the synchronization with different users can access simultaneously the service files creating mismatches between the information stored and the actual state of the synchronization. After having fetched the information about the user of the application (user name and identification code) and the ones about the state of the network connection (IP address, subnet mask and broadcast address) the process immediately forks giving birth to a new process. The parent will enter listening mode while the children will enter the search mode.

When in search mode, the process sends 10 UDP broadcast hello messages containing info about user name, identification code, IP address and local time and date of the computer. Every time an answer is received a new process is created and if the user we made contact with belongs to our friend is list and it is not already connected with us the synchronization process is started else a negative answer is sent back and the process is terminated. The previous two condition can be easily verified by checking the two service files *currentIpConn.txt* and *currentFriendsConn.txt* which are constantly kept up to date.

After the 10th hello message has been sent, and either an answer or none as been received, this process is terminated because it is assumed that all the users connected to the network should already have been found.

When in listening mode, the process keeps monitoring the specified socket for incoming UDP messages. As soon as a new message is received, a new process is created with the purpose of handling the request. Again, if the user the process made contact with belongs to the list of friend and is not already synchronizing its files with us, a positive answer is sent back. Else, no message is sent back and the process is terminated. The content of the answer to the hello message contains not only the info about user name, identification code and IP address, but also the number of the port on which the reliable TCP connection, used during the synchronization stage, will be established. The local time and date of the other pc is used to compute the so-called data-correction, which represent the delay in seconds between the times of the two machines. This variable guarantees a correct synchronization of the shared files even if the computers are not set on the same time, because it is going to be used during the comparison of the files owned by the users.

Differently from the process in search mode, the one in listening mode is never terminated, as it must be always ready to accept incoming messages from newly connected users.

When the two conditions needed for the synchronization stage to be started are satisfied, a reliable TCP connection is established between the two computers on the socket previously chosen by the process in listening mode and all the messages needed to compare the files and take the right decisions start to be exchanged. This is not enough though for the application to run correctly, because in order to be able to identify if the other user disconnected two more processes need to be created on each side. For every synchronization stream, two processes are needed with the purpose of keeping the synchronization alive; they are in fact inspired to the keep-alive protocols. One process behaves as a server and the other as a client and they start respectively to exchange periodic messages with the two processes on the other side, which behave as a client and a server. These messages are sent over two separated TCP connections established on the two successive ports of the synchronization stream. While the processes keep receiving messages from the other user, they know the connection is still on. As soon as one of these messages is not received before the relative timer elapses, the keep alive processes terminate firstly the synchronization process and secondly themselves. Thanks to this stratagem, the synchronization processes cannot remain stuck forever while waiting for a message coming from a process that does not exist anymore.

As already mentioned, the service files saved in Documents/Multibox/ are constantly kept up to date. When a synchronization stream is terminated by killing the liked processes, *userPid.txt*, *currentIpConn.txt* and *currentFriendsConn.txt* are immediately updated in order to mirror the state of the connections the application is currently managing.

Three service files not yet introduced are the databases. For the synchronization stage to run properly, a local database, a temporary database and a back-up database are necessary for every friend in the list of friends. As every user can be uniquely represented by its identification code, the local database is named *userDatabaseLocal-user code-.txt* (for example the local database relative to the friend Carlo with ID 1234567890 is going to be saved as *userDatabaseLocal1234567890.txt*). The same name structure is given to the temporary database, *userDatabaseTmp-user code-.txt*, and to the back-up database, *userDatabaseBackup-user code-.txt*. In these databases, the information about all the shared files is stored. An example is given here: e4c23762ed2823a27e62a64b95c024e7 /Users/Carlo/Sites/SharedFolder/SharedFile.mp4

1339772122.

Every line of the database saves the information about the hash of a file, its path and its last modification date calculated in seconds from January 1st 1970 00:00.

The local database saves a snapshot of the synchronization state every time a synchronization phase is completed. This means that two user will always have matching local databases. The temporary database saves a snapshot of the content of the shared folder every time a new synchronization phase is being started. Finally, the back-up database saves the information of the files yet to be synchronized and only the synchronization processes started from a listening-mode process use it.

The process that leads the synchronization is the one that has been generated by the process in search-mode. It is its responsibility to iteratively select all the files in its shared directory (which have been just listed in the temporary database) and send the relative information to the other user before getting the information about the files that the other user has added.

The process can enter in one of four different loops depending on the content of the local database and of the temporary database. If both the local and the temporary databases are empty, there are currently no files in the shared directory and there where no files also at the end of the last synchronization. This means that the user has no files to share with the other user and that he will only have the get the files from the other user, if any have been added since last synchronization.

If the local database is empty and the temporary database contains information about some files, the user added some files since last synchronization and the process needs to send the information about all the files to the other user. In addition all the new files added by the other user are going to be fetched.

If the local database contains information about some files and the temporary database is empty, the user deleted all the files he was sharing since last synchronization. In this case the objective of the process is to send the information about all the files in the local database, which the other user will have to delete. Again, all the new files added by the other user are going to be fetched.

Finally, if both local and temporary databases contain information about files, the two processes exchange first the information about the files to delete (identified by comparing the two databases) and secondly the information about the file that need to be updated and synchronized.

On the other side of the end-to-end TCP connection, the synchronization process generated by the process in listening-mode has the aim of comparing the information received from the other user with the one computed directly on the files saved in the shared directory.

In order to do this, a special function has been created. It enables the process to confront the information received about the target file with the information about all the files saved into the temporary database. For every target file, the temporary database is iteratively read.

If a file with the same hash of the file of the other user is found, the files on the two computers are the same version. Their names are immediately checked; if they match the files are up to date, else one of the two has been renamed. Local and temporary databases are immediately checked in order to discovered who has to rename his file.

If a file with the same name but a different hash code is found, it means that one of the users updated his file to a new version. It is easy to discover who made the update by checking the last modification date; the other user is notified that he needs to download the new version.

Finally if no matches can be found the file is missing and needs to be downloaded. The case in which a file has been both modified and renamed falls in this case; before the new version is

downloaded the old one is erased. Every time a file is downloaded, updated, renamed or delated the relative service file (added.txt, updated.txt, renamed.txt or deleted.txt) is modified adding a star to the first line. As mentioned earlier, the GUI periodically checks the content of these service files and notifies through Growl the events.

When all the files have been correctly synchronized, a new synchronization phase is initialized and the application keeps looping the operations which have been just described.

Errors can occur while handling the service files and while establishing the connection. Is such errors occur, a service file named *error.txt* is created and it is the GUI that steps in terminating the processes and re-starting the application.

We observed a mismatching behavior in the way the keep-alive processes respond to a disconnection of the other user or to an interruption of the synchronization. In this second case, Multibox is able to identify immediately the interruption of the service while in the first case it takes about 70 seconds to notice the absence of the other user. We are not able to define the difference between the two cases but this behavior is not going to create problems as a disconnection usually indicates that the user is leaving the working space.

Link for downloading MMNET_Team03_Sw.zip:
https://dl.dropbox.com/u/6151499/MMNET_Team03_Sw.zip.