Universidade Federal do Rio Grande do Norte
Instituto Metrópole Digital

Course: Information Technology
Subject: DIM615 - “Project of Operational Systems”
Professors: Ivanovitch Medeiros Dantas da Silva and Edgard de Faria Correa

Software Project Report

octo-sync: Application to synchronize files and folders across computer systems

by Pitágoras de Azevedo Alves Sobrinho

June 29 of 2017
Natal-RN, Brazil

1. Introduction

Let's say a user wants to have his project in two different computers, while he is writing it. But what to do if one file is changed in one of the computers, but these changes need to be on the other computer too? And if a directory is deleted at one computer, but note at the other? What this user needs is a way for the files and folders to be synchronized. Some options, on Linux:

- Copy everything to a flash drive or disk, insert this media on the other computer and copy the new version of the data, every time a change is made to synchronize;
  - Time expensive;
  - Needs external media;
  - Manually choose which folders and files to synchronize;
- Use a *cloud* software such as *Google Drive* or *Dropbox:*
  - Requires internet connection;
  - Limited storage space;
  - The user is litteraly giving away his precious files to some private company, who may or may not respect his privacy;
  - Some are not available for Linux Systems;
- Version Management systems such as *Git*:
  - High learning curve;
  - Not automatic: user needs to commit, pull, push...
  - Mostly for programmers, not very general purpose;
- Transfer with commands like *scp* and *cp*:
  - Writing the same (long) command every time to transfer something, after each change;
  - These commands just copy files, they do not delete files and folders too;

Given the disadvantages of these options, the *Octo-Sync* project was made to provide a different option for Linux users.

2. Description

Octo-Sync is a computer program capable of automatically synchronizing a folder in different computers, connected through a network. Once the user changes or creates files/folders in the folder being synchronized, *Octo-Sync* will automatically transfer the changes to the other computer.

Overview of how to use Octo-Sync:

1. Install it;

2. Open it on the folder you want to synchronize, to start hosting it;
3. Open it on the other computer, which will synchronize with the host;
4. The user can do what he wants inside the folder, Octo-Sync will take care of the rest;

2.1. Packages required:

- scp: Command to transfer files over a network using ssh encryption. Octo-Sync uses this command to do the actual file transfers;
- sshpass: Used to pass user password directly to scp, without prompting the user to write it;
- openssh-server: To receive files with scp from other machine, a machine needs to be running a ssh server, this package installs the openssh server;

2.2. Installation:

Octo-Sync is completely open-source, available on github and has a Makefile, so all the user needs to do in order to build it is:

```
$ git clone https://github.com/pentalpha/octo-sync.git

$ cd octo-sync

$ make
```

This will build the executable of octo-sync, that can be copied or linked to anywhere on the file system.

2.3. Usage instructions:

```
$ ./octo-sync host hostAddress=[local-ip-address] hostPasswd=[a password]
```

The host will wait for a connection from other computer on the network:

```
$ ./octo-sync sync hostAddress=[host-ip-address] hostPasswd=[same password from above]
```

The main security of the octo-sync host is its password. Only with the same password another instance of octo-sync can connect to it and synchronize files. There are more command line options for more detailed use, but these are the above are the mandatory ones. More detail on the arguments can be accessed with the -h argument, the output is the following:

```
$ ./octo-sync -h

sync or host

       The operation to perform: either host or sync;

       The host must be the first instance to be executed, it will wait for
a connection;

       The sync connects to a host that is waiting for a new connection;

Use [arg-name]=[value] to pass arguments:

* = obrigatory

syncDir

        The directory to synchronize. Default = ./;

hostAddress

       * Hosting machine ip;

hostPort

        Hosting machine port. Default = 50002;

localPasswd

        Local machine (sync or host) user password. The remote will use this
password on the scp command;

hostPasswd

       * Server password of the host. Used to login;

scpPort

       Specify a port for the scp command. If not specified, the default
will be used;

logLevel

       Minimum level of log messages. A lower value means more output.
Default = 7;
```

The user only needs to call octo-sync with a lot of arguments one time. The values of these arguments are stored on .octoConfig file, inside the synchronization folder. This file is ignored during the synchronization. If a different argument is used the next time octo-sync is called, the file is overwritten with the new values of the arguments.

3. Development

The project was developed using the C++11 language. The language was chosen keeping in mind that the main objective of the project is educational. C++ grants direct access to the system calls and the use and comprehension of them is one of the main objectives of the DIM615 subject. C++ is also a popular language that can be compiled on most environments. Also, the binary generated by the C++ compiler don't need anything but itself to run: no need to install an interpreter or other software like with Java or Python. The C++11 standard was chosen because it makes C++ a more modern language and offers many new and useful features. The compiler used was GCC 6.3.

The source code was written with the Visual Studio Code editor, an open-source code editor developed by Microsoft that enables C++ autocomplete and syntax highlighting. The only external library, not part of C++'s STL, used was tinydir (https://github.com/cxong/tinydir). It consists of a small header file to be used to iterate through directories and files. It was useful to scan folders recursively, listing the files contained. No debugging software was used, just the output of the logging system (logging.h) was already enough to diagnose errors.

3.1 Modules Overview

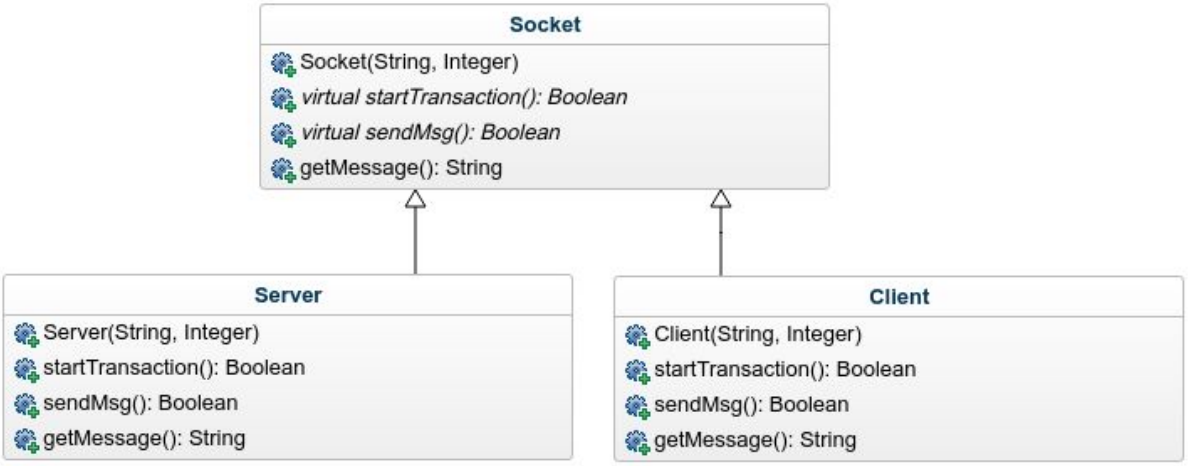
Building a synchronization application is not all about sending files. This is a list of the main modules in **octo-sync** and their purpose:

1. Generic purpose Posix **socket class**, to be implemented by **Server** and **Client** classes;
2. Module to automatically scan folder looking for new files, new folders modifications, deletions and to keep track of all this: **SyncDir**;
3. Command Line Interface: **ArgParser** and **OctoSyncArgs**;
4. A finite-state machine: **SyncBot;**

3.2 Sockets

I built a generic purpose socket that can send and receive messages, using a parallel thread to receive the messages and store them in a queue. The socket class is abstract and left the details on how to start a connection for the Server and Client children classes. This way, it was not necessary to write much server or client specific code.

If I had not done it that way, the project would have been much more complicated to write. The simplified class diagram is the following:



With these classes, octo-sync only has to store a pointer to the generic Socket class. This pointer can point to either a Server or Client and both are used the same way.

3.3. SyncDir

For a synchronization tool, being able to know what is happening with the files is essential. SyncDir is a class that monitors a folder. It keeps track of everything: files and subfolders. The files/folders are found using the tinydir library and the information about the last modification is given by the stat() system call.

But it only notifies the changes: modifications, creations and deletions. That is very important for the performance.

3.4. CLI interface

For programs that need to be runned remotely or on servers, a graphic interface is not always an option. Octo-Sync is used through a command line interface. It has different behaviour depending on the arguments given. These arguments are given on the form:

```
./octo-sync [operation] [arg1]=[value1] [arg2]=[value2] [arg3]=[value3]
```

There is no specific sequence for them and some are mandatory, others not. Some are mandatory, but have default values so they do not need to be specified every time, like syncDir and hostPort. Another feature is that Octo-Sync stores the last arguments on a file inside the synchronization directory. That means the user only has to type a lot of arguments once.

3.5. SyncBot

Before more explanations, I would like to make it clear that there are two main types of synchronization done by **octo-sync**:

- Metadata synchronization;
- Data synchronization;

These two synchronizations are coordinated by a class called SyncBot.

3.5.1 Metadata Synchronization

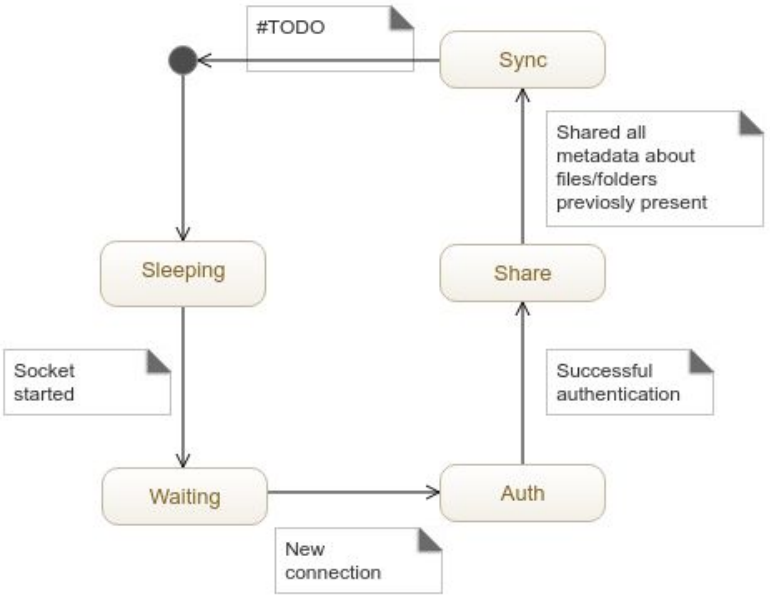
To know WHEN and IF synchronizing a file/folder is needed, the host or sync instance need to know not only their current files and the last changes: they need to know the same about the machine on the other side too. Both sides (host and sync) have two instances of SyncDir, one for itself and other for the other side. The data transfers are shared through the network, based on that data.

3.5.2 Data Synchronization

The files are transferred using the scp command. It receives the remote user password, the address, the port and the file to copy and copies it using ssh. But some of the data synchronization is not about sending and receiving files: there are also messages to create folders, delete folders and delete files remotely when it is needed.

3.5.3 States

There was no initial intention to make SyncBot a finite state machine, but while I was writing it the name started to make sense, because it eventually became a finite-state machine. Only one behaviour is not enough to make everything this class needs to do. It needed more, and more, and more different behaviours:



3.5.3.1. Sleeping State

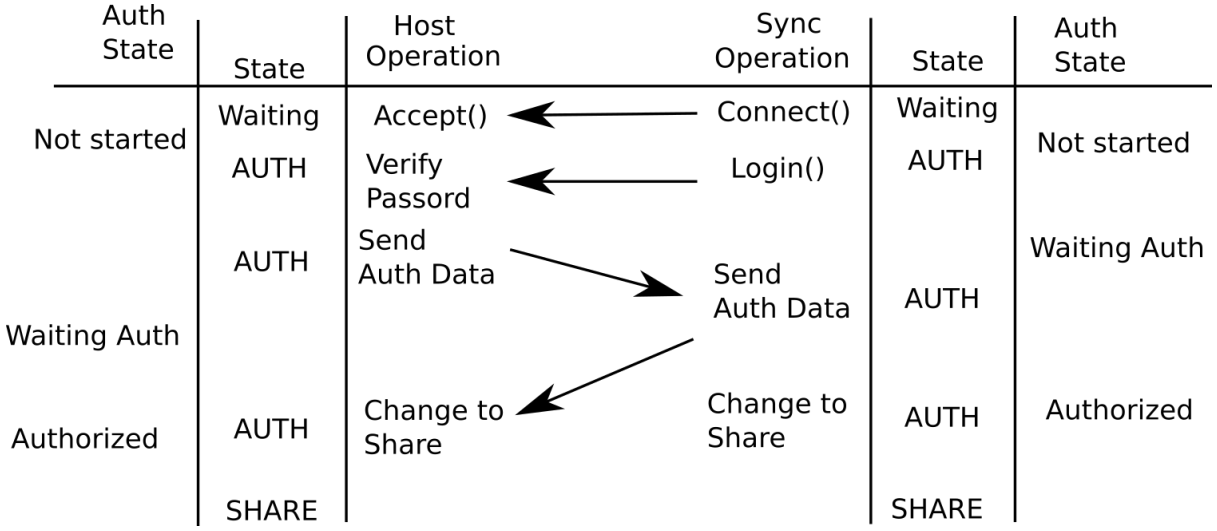
Initial state. The socket has not yet started, all the SyncBot does is updating SyncDir with the current files and folders. But, if the server/client is starting, it goes to the next state: Waiting.

3.5.3.2. Waiting State

Waiting for a new connection, all the SyncBot does is updating SyncDir with the current files and folders. But, if the server/client is connected to something, it goes to the next state: Auth.

3.5.3.3. Auth state

The objective of this stage is ensuring that the user credentials are not shared with basically anyone. A sync instance of octo-sync need to send the right password to the host. After that, each side (sync and host) send their credentials, preferred ssh ports and synchronization directories. This state has a few sub-states, that change with the progression of the communication with the other side:



3.5.3.4. Share State

The local SyncDir is updated, the other SyncBot has connected, logged in and sent his information about username and etc. Everything is ready? Not everything: The SyncDir that mirrors the local SyncDir on the other SyncBot has not been updated yet. There is no way to know if a file should be transferred or not.

The Share State is for sharing only file metadata, not data. Once all metadata has been shared, a “shared-all” message is sent to the other SyncBot and the next

stage starts: Sync.

3.5.3.5. Sync State

In this state, both data and metadata is synchronized. It is here that octo-sync will remain most of the time.

Before a synchronization, SyncDir is updated. Then, if there are changes on the change queue, a change is popped and SyncBot tries to send it. Before sending, it needs the permission of the other SyncBot, and waits for this permission.

Receiving and sending changes CANNOT be done at the same time, or crazy errors happen. That's why a mutex called syncLock is used.

```
342  void SyncBot::sync(){
343      log(1,"SYNC-BOT", string("Trying to lock on sync()"));
344      syncLock.lock();
345      log(1,"SYNC-BOT", string("Locking on sync()"));
346      updateLocalDirIfNotBusy();
347      if(localDir.hasChanges()){
348          SyncChange change = localDir.nextChange();
349          if(change.path != ""){
350              log(3, "SYNC-BOT", string("Going try sync: ")
351                  + change.path);
352              sendStartSync();
353              while(syncAllowState == WAIT){
354
355              }
356              if(syncAllowState == ALLOWED){
357                  sendChangeInfo(change);
358                  sendChange(change);
359                  //sendChangeMsg(change);
360                  localDir.popNextChange();
361              }
362              syncAllowState = WAIT;
363              sendEndSync();
364          }
365      }
366      log(1,"SYNC-BOT", string("Unlocking on sync()"));
367      syncLock.unlock();
368  }
```

4. Conclusion

This software is currently capable of synchronizing files and folders over a local network. Until this point, it has not yet been tested on huge workloads: folders with hundreds of subfiles and large files with more than 100MB.

Not everything that could be done was done, because the time was very short. There are a few features that octo-sync could have:

- **Gracefully exiting**: Octo-Sync can only be ended with a SigInt or SIGKILL. There is no other easy way to finish it than pressing CTRL+C, killing the process, or disconnecting the other syncbot. Another way is needed to be implemented.
- **Complete Cycle**: The Sync State is perpetuous. It would be nice to, instead of crashing the program when the other SyncBot disconnects, go back to the sleeping state and wait for a new connection;
- **Trash**: Once a file is deleted with octo-sync, there is no going back. A trash folder would make the program more safe.
- **stop using user password**: sending the password of the user over the network is not safe at all. Future versions of octo-sync will use ssh keys.
- **Multiple machines connected to host**: why is it called octo-sync, from octopus (animal with many members), if only two sides can synchronize with it? Well… i also plan to do it on the future.