CS 455/555: Programming Project 1
Simple File Synchronization Server with Sockets
100 points
Due date on class home page

From: http://en.wikipedia.org/wiki/Servent
Servent
From Wikipedia, the free encyclopedia

In general a servent is a peer-to-peer network node, which has the functionalities of both a server and a client. This is a portmanteau derived from the terms server and client, and is a play on the word "servant". The setup is designed so that each node can upload, download, and usually also route network information, allowing for the creation and maintenance of ad-hoc networks.

The term originated from the Gnutella lexicon, with Gnutella being the first widespread decentralized peer-to-peer network.

1 Introduction

In this project, we will write a simple file synchronization **servent** (server+client). The servent runs on two machines: on one machine in server mode and in the client mode on the other machine. For example, we would run the servent in server mode on the machine that we are using actively and in client mode on the our other machine that is not being used currently. The idea is that in the server mode, the servent watches a specified folder for any changes in the files. Any files that change trigger corresponding actions on the client. As a result the folders on the two machines remain synchronized. For simplicity, we will assume that the two folders to be synchronized do not contain any subfolders.

2 Details

- Please name the main servent class be *FileSync*. It may be invoked in two different ways:

```
java FileSync -client <serverhost> <localfolder>
java FileSync -server <folder>
```

- The config directory for the servent will be a hidden folder .fss in the directory the server is running from. That folder contains a config file named fssrc, which contains a list of keywords followed by the equals sign and then one or more values. The keyword clientlist gives a comma separated list of Internet hostnames from which connections will be accepted. If this line is missing or contains no hostnames, then the server only accepts connections locally. The keyword interval gives the synchronization interval, which has a default value of 60 seconds. Next, the logfile keyword specifies the full path of the log file that records all the synchronization actions for both clients and servers.

```
clientlist=host1,host2
interval=60
logfile=~/.fss/log
timeout=900
```

- In the client mode, the servent connects to the server mode servent. Initially, the client sends its current list of files with their sizes and time stamps. The server would respond back with its list of differences, that is, file names that need to be copied back. The client then requests each file from the server. The server also needs to let the client know if files get added or deleted.
- You will devise a simple object-based protocol for the client/server to communicate. Issues that you will have to consider include if the files are transferred in one chunk or in smaller chunks, how to communicate errors like “permission denied", “file does not exist", "not enough space" etc.
- Please make sure that you kill your servents before logging out of any lab machines.
- The servents should self-destruct is there is no activity for 15 minutes.
- In this project, we will use sockets to implement the filesync servent.

3 Multi-threaded Server

- Allow synchronized folders to contain subfolders. (10 points)
- (**Required for graduate students, extra credit for undergraduates**) Make the server be multi-threaded, that is, it should respond to a valid client by creating a new thread that synchronizes with the connecting client. However, the server should not serve more than 4 clients simultaneously. (10 points)

4 Documentation (20 points)

- Create a MPEG video that demonstrates various features and scenarios for the filesync servent.
- The README.md file must have at least the following elements:
  - Project number and title, team members, course number and title, semester and year.
  - A file/folder manifest to guide reading through the code.
  - A section on building and running the server/clients.
  - A section on how you tested it.
  - A section on observations/reflection on your development process and the roles of each team members.

5 Required Files

- Manage your project using your team git repository via backpack. Please name the project folder as p1.
- There must be a file named `README.md` with contents as described earlier in the Documentation section.
- The project must have a `Makefile`. You can use any build system underneath (like gradle, maven, ant etc) but the Makefile should trigger it to generate the project.
- All you source code, build files et al.

6 Submitting the Project

Open up a terminal or console and navigate to the backpack folder for the class. Navigate to the subdirectory that contains the source files that you worked on for the project/homework. Suppose that we are submitting three files `README.md`, `main.c` and `utility.c` and one folder named `include`.

- Clean up your directory
  `make clean`
- Remove any other unnecessary files that were generated by you or the auto-grader.
- Add all your changes to Git (this would be specific to your project)
  `git add main.c utility.c include/ README.md`
- Commit your changes to Git
  `git commit -a -m "Finished project p1"`
- Create a new branch for you code
  `git branch p1_branch`
- Switch to working with this new branch
  `git checkout p1_branch`

  (You can do both steps in one command with `git checkout -b p1_branch`)
- Push your files to the Backpack server
  `git push origin p1_branch`
- Switch back to the master branch
  `git checkout master`
- Here is an example workflow for submitting your project. Replace p1 with the appropriate project name (if needed) in the example below.

```
[amit@localhost p1(master) ]$ git checkout -b p1_branch
Switched to a new branch 'p1_branch'

[amit@localhost p1(p1_branch) ]$ git push origin p1_branch
Total 0 (delta 0), reused 0 (delta 0)
To git@nullptr.boisestate.edu:amit
 * [new branch]      p1_branch -> p1_branch

[amit@localhost p1(p1_branch) ]$ git checkout master
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.
```

- Help! I want to submit my code again! No problem just checkout the branch and hack away!

```
[amit@localhost p1(master) ]$ git branch -r
  origin/HEAD -> origin/master
  origin/master
  origin/p1_branch

[amit@localhost p1(master) ]$ git checkout p1_branch
Branch p1_branch set up to track remote branch p1_branch from origin.
Switched to a new branch 'p1_branch'

[amit@localhost p1(p1_branch) ]$ touch foo.txt

[amit@localhost p1(p1_branch) ]$ git add foo.txt

[amit@localhost p1(p1_branch) ]$ git commit -am "Adding change to branch"
[p1_branch 1e32709] Adding change to branch
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 foo.txt

[amit@localhost p1(p1_branch)âĘŚ ]$ git push origin p1_branch
Counting objects: 3, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 286 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To git@nullptr.boisestate.edu:amit
   36139d1..1e32709  p1_branch -> p1_branch

[amit@localhost amit(p1_branch) ]$ git checkout master
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.
[amit@localhost amit(master) ]$
```

- We highly recommend reading the section on branches from the Git book here: Git Branches in a Nutshell