P2P File Sharing Application (P2P-FS)
System Functional Specification

CMP2204 Term Project Spring 2020

1 Introduction

1.1 System Purpose

P2P File Sharing application consists of Service_Listener, Service_Announcer, P2P_Downloader, P2P_Server components. The purpose of the former two components is to automatically detect all available users in the network for downloading content from. The purpose of the latter two is to exchange files between two remote hosts.

1.2 Definitions, Acronyms, and Abbreviations

Throughout this document, the terms in boldface below are to be interpreted as defined:

shall	This term indicates an obligatory requirement that must be met to comply with the specification.
may	This term indicates an item that is truly optional.

1.3 Operational Scenarios

The following are the use cases supported by the P2P File Sharing Application:

Service Discovery: Upon connecting to the Local Area Network, our application starts listening for all P2P File Sharing services in the LAN. Each detected user and which portions of which files they have, are stored in a local dictionary.

Service Announcement: Upon connecting to the Local Area Network, every peer starts to periodically broadcast the list of all files they have. As they have more files, they start announcing more content.

Downloading a file: The end user specifies a content to download, and as many TCP sessions as needed are opened, with the users that hold subparts of the requested file, to download all subparts of that file. When a download of a subpart is complete, TCP session is closed.

Serving (parts of) files: Every user has one original file, which, they will divide into N-byte chunks. In addition, every user will altruistically serve all pieces of other users' files that they downloaded. When a new TCP connection request is received, P2P_Server immediately accepts this connection request, receives and parses the message, and sends the requested file to downloader.

Download history: User can view the download history (date/time, from which user, content name, part index).

2 Requirements

2.1 Service Discovery Requirements

Req. #	Requirement
2.1.0-A	When launched, Service_Announcer **shall** first ask the user for its username and store this name locally.
2.1.0-B	Service_Announcer **shall** periodically send broadcast UDP messages announcing its service. The period **shall** be once per minute.
2.1.0-C	Service_Announcer's periodic broadcasts **shall** contain a JSON that contains the username and the list of hosted files. It is very important that the field names are named "username" and "files" and the format is a valid JSON format; otherwise you may have parsing issues when working with your peers. An example would look like: '{"username": "Ece", "files": ["ece_1", "ece_2", "ece_3", "ali_2", "ayse_1", "cem_1", "cem_2"]}'.
2.1.0-D	Service_Announcer **shall** be able to read the files names under a specified directory, and insert them into the message in JSON array format. For this, Service_Announcer **shall** be launched *after* P2P_Server (after the P2P_Server prepares the file chunks for serving).
2.1.0-E	Service_Listener **shall** listen for UDP broadcast messages on port 5000. The broadcast IP address should be configured programmatically, by replacing the last (4^{th}) decimal number with 255 in the host IP address. For example, if your IP address is 192.168.2.34, you should broadcast to IP address 192.168.2.255.
2.1.0-F	Upon receiving a message, Service_Listener **shall**: *(i)* parse the message contents using a JSON parser in Python, *(ii)* get the UDP broadcast sender's IP address using recvfrom() method.
2.1.0-G	Service_Listener **may** display each detected user and their served content on the console (*e.g.,* "Ece : vid_1, vid_2, vid_3"). This would also help you with debugging the listener code.
2.1.0-H	Service_Listener **shall** store the list of files (parsed from the JSON message) in a dictionary. Let's call this *the content dictionary*. The dictionary keys **shall** be the *content chunk name* (*e.g.,* ece_1) and the value **shall** be an array containing the list of IP addresses having that chunk (that you fetched using recvfrom()). This dictionary **shall** be shared with P2P_Downloader process. You **may** store it in a local text file that is shared between the Service_Listener and P2P_Downloader components.

2.2 Download Requirements

Req. #	Requirement
2.2.0-A	When launched, P2P_Server **shall** ask the user to specify the file it will initially host (*i.e.,* that user's original file.) P2P_Server **shall** divide the specified file into N-byte (e.g. 200-byte) chunks and store them as separate files with indexed naming. (The code for this will be provided to you.) Once this is done, P2P_Server **shall** display a message on terminal, stating it is ready to host these files. To simplify the design, let's assume each file in our system always has 5 chunks.
2.2.0-B	P2P_Server **shall** listen for TCP connections on port 5001.
2.2.0-C	P2P_Server **shall** accept TCP connection request before it times out, and **shall** successfully send the content requested by the process at the other end.
2.2.0-D	P2P_Server **shall** parse the JSON in the message to learn which part of which file is being requested by the sender, and it **shall** send this file to the requester over the TCP connection.
2.2.0-E	When launched, P2P_Downloader **shall** prompt the user to specify which content it wants to download. For the user-entered filename, P2P_Downloader **shall** initiate **5** sequential download procedures for each chunk of this file, as described in the following requirements, in order to download all chunks of that file.
2.2.0-F	P2P_Downloader **shall** lookup its *content dictionary* to fetch the list of users (*i.e.,* IP addresses) having a certain chunk. For this, it'll lookup the dictionary with the key set to *specified_chunk_name*+"_" + *index*, for *index* values of 1 through 5 (*e.g.* ece_1, ece_2, ece_3, ece_4, ece_5).
2.2.0-G	P2P_Downloader **shall** lookup its *content dictionary*, which requires a lookup to the local file that Service_Listener wrote the content dictionary into. For downloading a chunk, P2P_Downloader **shall** try downloading this file from the first user in the array that is in the *content dictionary* for this chunk name. If download is successful, P2P_Downloader **shall** move on to the next chunk. If it is not successful, P2P_Downloader **shall** try downloading from the other users in the array until download of that chunk is successful. If all users in array have been tried and that chunk cannot be downloaded, P2P_Downloader **shall** display a warning message (on the console) to the user informing about the problem. (*e.g.,* "CHUNK ece_1 CANNOT BE DOWNLOADED FROM ONLINE PEERS.")
2.2.0-H	For downloading each chunk, P2P_Downloader **shall** initiate a TCP session with the specified user's (random user hosting that chunk) IP address. The message **shall** contain a JSON that has a key of "filename" and value of the name of chunk to be downloaded. Example looks like: {"filename": "ece_1" } .
2.2.0-I	P2P_Downloader **shall** close a TCP session upon receiving the chunk it requested.
2.2.0-J	Without running a validation on the downloaded content, we'll assume the file has been correctly downloaded when all 5 chunks have been downloaded. After the 5^{th} chunk has been downloaded, P2P_Downloader **shall** combine these 5 chunks into a single file. (I'll provide the code for this, which you'll integrate in your P2P_Downloader code.) Once the file is ready, the P2P_Downloader **shall** inform the user via the terminal that the file has been successfully downloaded.
2.2.0-K	P2P_Server **shall** dump all served filenames in a Server log (a text file) under the same directory. Each entry **shall** specify timestamp, sent_to_IP_address, sent_chunk_name.
2.2.0-L	P2P_Downloader **shall** also dump all downloaded filenames in a Download log (a text file) under the same directory. One entry **shall** contain timestamp, chunk_name, downloaded_from_IP_address.
2.2.0-M	After a TCP session is closed, P2P_Server and P2P_Downloader **shall** persist; the service **shall not** terminate.

2.3 Performance Requirements

Req. #	**Requirement**
2.3.0-A	P2P-FS **shall** run on Python 3.
2.3.0-B	Service Listener **shall** be able to detect all online users.
2.3.0-C	Content dictionary **shall** be able to keep up to 10 users' contents, with each having up to 10 chunks, with each chunk carried by at most all other users.
2.3.0-D	P2P_Downloader **shall** be able to download a file from any online user, with no perceivable delay.
2.3.0-E	Any unspecified configuration is a plus – displaying download progress, displaying error message when file chunks can't be downloaded from peers, etc.