Overview

This project is a simple peer-to-peer (P2P) protocol implementation with distributed index (DI) using socket programming. The project in its current implementation is used to share RFCs, however it can be extended to share files of other formats as well.

The code is written in Python 2.7 and uses standard python libraries. Basic configuration of the server and peers are maintained in a JSON file. The implementation divides the code into four modules as follows :

1. peer
2. rserver
3. records
4. utils

The repository also includes an rfc directory that contain RFC files with a metadata file to initialise the index of the peer on startup. Please see the readme.md file in the project for instructions to run the application.

IMPLEMENTATION

Registration Server

The registration server is implemented in the rserver module. The server is configured using config.json file within the project's directory which instructs the server to listen on a particular port and hostname/ipaddress. When a peer tries to connect to the registration server, four types of requests are accepted by the server:

1. Register
2. Leave
3. PQuery
4. KeepAlive

These are the commands that invokes over a TCP connection with appropriate message format (discussed shortly).

Using the records module, the server also maintain a list of peers that have registered with it in the form of a linked list. Each node contains an object of class implementation that has following attributes :

Attribute	**Type**	**Description**
hostname	string	Hostname of the peer
port	string	Port of the peer
cookie	integer	Cookie assigned to the peer by RS
active	boolean	State of the peer, calculated using ttl value
ttl	integer	TTL of the peer (Max. 7200 seconds). This value is auto-decremented in a timely manner.
reg_count	integer	Number of times the peer registered/contacted RS
last_reg	datetime	Time when the peer was last registered

In addition to that, the class contains the implementation to do housekeeping tasks on these nodes such as updating ttl, updating registration count, marking inactive etc.

Peer

The peer is implemented in the peer module which contains both the server and client implementation of the peer. The peer is configured using config.json file within the project's directory which instructs the peer server to listen on a particular port. This also informs the client of the registration server port and hostname. We further discuss the server and client roles of the peer individually:

As client

The role of the client process is to interact with the user and make requests on user's behalf to the registration server and other peers which may act as servers of RFCs. Four types of requests are supported by a peer when communicating with registration server:

Command	**Description**
Register	1. Registers the peer with a RS server. 2. Receives a cookie in the response 3. Updates the TTL on server to 7200 if request is sent again
Leave	1. Peer indicates the server that it is leaving the swarm 2. Server removes the peer's cookie and marking it unregistered
PQuery	1. Peer queries the server for a list that contain hostname and port of other peers in the swarm 2. Receives a list of hosts with their server ports which the peer uses to overwrite its existing list (if any) 3. Updates the TTL to 7200
KeepAlive	1. Updates the TTL to 7200

Following commands are supported when two peers communicate:

Command	**Description**
RFCQuery	Queries a peer for its RFC index and merges it with the local RFC index
GetRFC	Requests an RFC from a peer

These are the commands that invokes over a TCP connection with appropriate message format (discussed shortly).

The peer stores following 3 important objects as its 'current state':

1. Cookie assigned by server

2. List of known peers as retrieved from the server
3. Pointer to the head of a linked list that stores the RFC index information

Using the records module, each peer maintains information about RFCs in an index locally. This is populated initially using the metadata.json which contains the information about local RFCs and after an RFCQuery command is executed, the index is updated by merging the local index with the index from peer appropriately. The metadata.json is automatically updated if and when an RFC is downloaded. This data structure is implemented in the form of a linked list where each node contains an object of class implementation that has following attributes :

Attribute	**Type**	**Description**
hostname	string	Hostname of the peer where this RFC is present
number	integer	Number of an RFC
title	string	Title of the RFC
ttl	integer	TTL of the record (Max. 7200 seconds). RFCs locally available have the max TTL. TTL is auto-decremented for other records.

In addition to that, the class contains the implementation to do housekeeping tasks on these nodes such as updating ttl, updating hostname after downloading etc.

As server

The server is implemented as a thread that runs in the background and listens on a port configured in the config.json and subsequently advertised to the registration server. The server supports two types of requests based on the type of command in it as discussed above:

1. GetRFC
2. RFCQuery

A single socket is opened for serving the entire flow of a command and closed after the request has been served. This is also true when an RFC is transmitted from one peer to another. The only difference is that while sending an RFC, multiple messages would be sent over the same socket. Of these messages, the first one would contain the RFC found response along with the format of the file being shared. The subsequent messages would be stream of binary data with each message of size 1024. The file is transmitted until the end of file (EOF) is reached.

MESSAGE FORMAT

The records module implements the representation, storage, serialisation and deserialisation of the messages. Same format is used during peer to peer and peer to registration server communication. Using the pickle library they are serialised while sending and deserialized while receiving. This is done to ensure that the objects are reconstructed when they are received. The messages exchanged are stored in form a request or response object as described below.

Request message format

Command	Version
Hostname	OS_version
Data	

Response message format

Status Code	Version
Hostname	OS_version
Data	

The data field is a dictionary object. For each of the allowed commands, this dictionary might contain zero, one or multiple keys that are of significance for the command. In addition to that, we also describe the following four status codes that any response message will contain.

Status Code interpretations

Status Code	**Interpretation in P2P protocol**
100	Resource Not Found
200	Resource Found
201	Resource Updated
300	Bad Request

Examples

Requests	Responses
Register Sequence	

Register P2P-DI/1.0
Laptop_1337 Linux_4.10.0-33-generic
{
'cookie': None,
'port': 65401
}
	200 P2P-DI/1.0
	Laptop_1337 Linux_4.10.0-33-generic
	{
	'cookie': 1
	}

PQuery Sequence	

PQuery P2P-DI/1.0
Laptop_1337 Linux_4.10.0-33-generic
{
'cookie': 2,
'port': 65402
}
	200 P2P-DI/1.0
	Laptop_1337 Linux_4.10.0-33-generic
	{
	'peers': [{
	'hostname': 'Laptop_1337',
	'port': 65401
	}]
	}

KeepAlive Sequence	

KeepAlive P2P-DI/1.0
Laptop_1337 Linux_4.10.0-33-generic
{
'cookie': 2,
'port': 65402
}
	201 P2P-DI/1.0
	Laptop_1337 Linux_4.10.0-33-generic
	{
	'message': 'TTL updated to 7200 seconds.'
	}

Leave Sequence	

Leave P2P-DI/1.0
Laptop_1337 Linux_4.10.0-33-generic
{
'cookie': 2,
'port': 65402
}
	201 P2P-DI/1.0
	Laptop_1337 Linux_4.10.0-33-generic
	{
	'message': 'Peer unregistered from the server'
	}

RFCQuery Sequence	

RFCQuery P2P-DI/1.0
Laptop_1337 Linux_4.10.0-33-generic
{}
	200 P2P-DI/1.0
	Laptop_1337 Linux_4.10.0-33-generic
	{
	'rfcs': [{
	'number': u '959',
	'title': u 'FILE TRANSFER PROTOCOL (FTP)'
	}, {
	'number': u '1945',
	'title': u 'Hypertext Transfer Protocol -- HTTP/1.0'
	}, {
	'number': u '2616',
	'title': u 'Hypertext Transfer Protocol -- HTTP/1.1'
	}, {
	'number': u '4632',
	'title': u 'Classless Inter-domain Routing (CIDR): The Internet Address Assignment and Aggregation Plan'
	}, {
	'number': u '5905',
	'title': u 'Network Time Protocol Version 4: Protocol and Algorithms Specification'
	}]
	}

GetRFC Sequence	

GetRFC P2P-DI/1.0
Laptop_1337 Linux_4.10.0-33-generic
{
'rfc': u '959'
}
	200 P2P-DI/1.0
	Laptop_1337 Linux_4.10.0-33-generic
	{
	'format': u 'txt'
	}
	
	<RFC file chunk size 1024>
	.
	.
	.
	.
	.
	EOF