Université Paris Sorbonne Nord
Institut Galilée

Master 1.
Distributed Java Programming
Project

1 Introduction

In a peer-to-peer (P2P) network, different from the traditional client-server architecture, all the nodes have the same status. That is, there is no a clear distinction between a client (that usually sends requests) and a dedicated server (that usually provides services for the clients). Hence, instead of writing a specialized code for the server and a completely different code for the client, in the P2P model the same code runs on all the nodes of the network that act as both, clients (when sending a request) or servers (when answering a request from another node).
This kind of architecture offers some benefits. In particular, since it is completely decentralized, it is more resilient: if some (or even several) nodes fail, the network can still work[1]. It is also possible to add more nodes and resources on the fly, thus augmenting the capabilities of the network. More information about P2P networks can be found for instance here and some legal, ethical and security issues related to P2P networks here.

2 Project Description

In this project, we will implement a P2P service for file sharing with some particular requirements. The idea is that each client/peer makes available some files that the other peers can download. In the following, we describe the expected behavior of the system.

Messages

The peers in the network interact by sending messages to other peers. As stated before, each peer executes exactly the same code since there is no distinction between the nodes of the network.

[1]Note that in the client-server model, if the server fails, there is no service at all!

Your system must support the following messages/actions:

- **name?** When a peer B receives this message from another peer A, B must send to A the pseudonym (nickname) she/he is using in the system. This can be useful in the interface of the system to better display the other peers.

- **known peers?** When B receives this message from A, it must answer by sending the list of peers that B knows. [2]

- **it's me!** A peer A may sent this message along with its IP and port to another peer B. Then, B adds A to its list of known peers and sends an acknowledge message to A. Depending on the configuration of the client, it can be the case that the size of the list of known peers in B is limited. Hence, if the list of known peers in B is already full, B sends an `error` message to A notifying that it could not be added.

- **file?** A peer A sends this message to a peer B along with a string s and an integer i when A is looking for a file whose name matches the string s. This message is completely asynchronous: A does not expect an immediate response from B since it might be the case that the file is difficult to find. Moreover, it could be the case that the response to this query comes from a different peer! More precisely, when B receives the message, it searches in its local files if one or more of them match the search criterion s. If this is the case, N sends a **voilà** message (see below). Otherwise, if $i > 0$, B resends the query to its list of known peers with the hope that some of them may have the file requested by A. When B resents the query, it decreases the value of i. This simple mechanisms is used to avoid that the **file?** message bounces indefinitely in the network.

- **voilà!** This message is used to notify a peer the files that match a given query.

- **download!** A peer A sends this message to a peer B to request the content of a file. Note that peers in the network can be disconnected at any time. Hence, the system must be able to try to reestablish the connection with B if the download fails at some point. If B does not respond after some time, A may try to find another peer to restart the downloading process.

- **bye!** This message is sent to notify that a peer is not longer willing to be part of the network. When a peer receives this message, it deletes the requester from its list of known peers and resends the message to its list of of known peers [3].

[2] This is one of the salient features of a P2P network. A new peer needs to know, at the beginning, only the IP and Port of only one peer already in the system. Then, it can discover and access more peers by adding new entries to its list of known peers.

[3] A mechanism similar to the counter used in the **file?** message is needed to avoid that the **bye!** message keeps bouncing indefinitely.

Deliverables and Rules

The project can be developed in pairs. Only one person per team must submit on Moodle a Zip file containing:

1. The source code, fully documented using Javadoc. The system must have a minimal interface (textual or graphical) to interact with it.
2. A detailed README file explaining how to compile and use the system.
3. A PDF document in the IEEE format of maximum 2 pages explaining how the system was implemented, the description of the main classes and how the problems related to concurrency and synchronization were handled.

Besides the requirements described above, it would be interesting to implement at least one of the following features. In that case, please mention it in the document:

- Configuration and relevant information for each client must be stored in a light (local) database, for instance, SQLite.
- Add to the interface an option to track the files that are currently being downloaded from the local repository of the client. This interface should allow also to stop sharing a particular file (thus disconnecting the peers that are currently downloading that file).
- Implement the communication between peers using secure sockets.
- In its simplest form, a peer download a file from a unique given peer. Hence, the total time to complete the download depends on the connection speed between the two peers. In order to improve the download process, allow a client to download simultaneously the same file from different sources.
- If the download of a file fails, the peer does not need to start the download from scratch. Instead, it can inform the amount of bytes already downloaded and start downloading from the point where the communication was interrupted.