ILLINOIS INSTITUTE OF TECHNOLOGY

CS 550: Advanced Operating Systems

Work realized by

Florentin Bekier
Rémi Blaise

P2P File Sharing System: Design Document

Taught by
Dr. Zhiling Lan

Master of Computer Science, Spring 2020

Contents

1 General architecture

P2P File Sharing System is made of 2 software:

1. The **Index** centralizes an index of all available files in the network. Each running instance of the Index creates a new independent file-sharing network.
2. The **Peer** can download and serve files from/to the network. Each running instance is a node of the network of the one Index it is connected to.

Peers serving files are identified by a unique UUID in the network, while files are identified by a unique ID made of the SHA-1 hash of their content and their name. This allows 2 files to be considered identical if they have the same content and the same name, and thereby to be distributed by several peers.

To distribute the charge more fairly between all the peers serving the same file, the Index always returns the list of peers in random order (see section 4.1.2). The client-Peer will then try to download the file from the first peer returned by the Index. If the download isn't successful, it will then try the next one and so on until one download request is successful or there is no more peer. After the download of a file, a peer automatically becomes a server for this file.

Both the Index and the Peer are designed to be easy to install and to configure.

2 Technology specifications

The 2 software are built using the following technologies:

- The Javascript (ES6) language interpreted by Node v12. Given this project is network-oriented and has low-performance requirements, this language is ideal for its very high-level community-built libraries, making the development very straight-forward, as well as its asynchronous-oriented design, making the servers fully parallelized, able to handle multiple requests at the same time. Because both team members have a strong previous knowledge of this technology, this is an obvious choice.
- The SQLite 3 database system handled by the Sequelize Object-Relational Mapper. This database system is selected for its simplicity of installation, the downside of this choice is not to have a querying interface as powerful as a traditional database system. Given the requirements of the project, the simplicity of the available SQL queries is not a problem.

Additional used tools:

- Git version control manager hosted on the GitLab platform.
- Node Package Manager (npm).

The 2 software are made by following the usual Javascript coding style and software design conventions.

3 Communication protocol

Our software can communicate in two different modes: **Index-Peer** or **Peer-Peer**. In both cases, the communication protocol is built on top of the TCP layer. The client and server communicate in a client-pull only matter by sending JSON-formatted data through a socket. The protocol is designed to enable the server to expose an **API** made of **procedures**.

In the sections below, we explain the communication process in detail for each mode.

3.1 Index-Peer

1. The Peer opens the connection over TCP.
2. The Peer emits a request in the form of a JSON message (see section 3.3).
3. The Index replies with a Response JSON Message (see section 3.4).
4. The Index closes the connection.

3.2 Peer-Peer

In a Peer-Peer communication, one of the peers will be the client and the other one will be the server.

1. The client opens the connection over TCP.
2. The client emits the request in the form of a JSON message (see section 3.3).
3. The server replies with a JSON message (see section 3.4).
4. The server streams the file through the socket.
5. The server closes the connection.

3.3 Request message format

```
{
    "name": "Procedure name",
    "parameters": {
        ...
    }
}
```

3.4 Response message format

Success message:

```
{
    "status": "success",
    "data": ...
}
```

Error message:

```
{
    "status": "error",
    "message": "Error description"
}
```

4 Design of the Index

The Index is built with the following software design:

- The `app.js` file is the entry point of the software and instanciates the TCP server.
- The `procedures.js` file defines the actual procedures provided by the Index API.
- The `repository.js` file handles the persistence layer by communicating with the database through the Sequelize ORM.
- The `config.js` file reads the configuration.

4.1 API

Using the previously defined protocol, the Index provides the following API:

4.1.1 The `registry` procedure

Enable a peer to register as a *server* in the network, or to update its server information.

A server information is made of:

- its UUID, chosen by the peer on the first start,
- its contact address (IP and port),
- its file list,
- the SHA-256 signature of the request certifying the identity of the peer,
- its public key which is required for first registration. In later requests, this parameter is optional and should only be added for updating the key. Note that the request still needs to be signed with the previous registered public key.

Expected request:

```
{
    "name": "registry",
    "parameters": {
        "uuid": "string",
        "ip": "string",
        "port": integer,
        "files": [
            {
                "hash": "SHA-1(content)",
                "name": "string",
                "size": integer
            },
            ...
        ],
        "signature": "string",
        "publicKey": "string"
    }
}
```

Expected response:

```
{
    "status": "success",
    "data": null
}
```

4.1.2 The `search` procedure

Enable a peer to search for a file by name in the network. Return the list of found results whose name is containing the given file name, with the lists of the peers delivering the file in random order.

Expected request:

```
{
    "name": "search",
    "parameters": {
        "fileName": "string"
    }
}
```

Expected response:

```
{
    "status": "success",
    "data": [
        {
            "id": "{hash}-{name}",
            "hash": "string",
            "name": "string",
            "size": integer,
            "peers": [
                {
                    "uuid": "string",
                    "ip": "string",
                    "port": integer
                },
                ...
            ]
        },
        ...
    ]
}
```

5 Design of the Peer

The Peer software follows the given design:

- The `app.js` file is the entry point of the software and starts the Peer client and the Peer server.
- The `client.js` file implements the client side: it registers the Peer to the Index, watches the shared folder and shows the CLI.
- The `server.js` file implements the server side: it instanciates the TCP server and listens for file requests.
- The `interface.js` file implements the communication protocol.
- The `files.js` file handles the file sharing management.
- The `config.js` file reads and initializes the configuration.

5.1 API

Using the previously defined protocol, the Peer provides the following API:

5.1.1 The `retrieve` procedure

Enable another peer to download a specific file. Return the file information and data as a stream.

Expected request:

```
{
    "name": "retrieve",
    "parameters": {
        "fileId": "string"
    }
}
```

Expected response:

```
{
    "status": "success",
    "data": {
        "filename": "string"
    }
}
```

5.2 Command-line interface

The user of the Peer software has access to the following actions:

1. List the current shared files served to the network.
2. Search by name for a file to download. List all corresponding files with ID and size. The user can then select the file he wants to download or return to the menu.
3. Exit the program.

To choose an action, the user enters the number corresponding to that action using its keyboard and press Enter to validate.

6 Security design

From a security point of view, the system could be vulnerable to the following attacks:

1. A malicious tier sends faulty request data to make the server-Peer or the Index crash.
2. A man-in-the-middle corrupts the file shared by the server-Peer to the client-Peer.
3. A malicious tier registers erroneous server-Peer information to the Index.

To address these vulnerabilities, we carefully designed the following defenses:

1. Validate any data received from a client so it doesn't make the server crash.
2. Identify the files by their SHA-1 hash so the client-Peer can verify the file is not corrupted after downloading it. If the hash doesn't match, the file is deleted.
3. Provide a public key at the first registration and cryptographically sign any further call to the `register` procedure.

7 Possible improvements

To go further, the following improvements could be brought to our P2P system:

- The Index, as is, doesn't support scaling given that Index data is centralized in a single database. An improvement could be to make it able to be installed on a distributed system.
- A GUI could replace the current CLI of the Peer, making it usable by common users.
- The downloading of big files could use several server-Peers if available. Similarly, the index could sort available server-Peers to better distribute the global charge over all servers.
- The Index could include a web-browsable interface listing available files on the system.
- The Index could reduce the risk of trying to contact an unavailable file server by periodically checking the availability of hosts or removing servers from the list after a given timeout.
- The protocol doesn't implement any incentive for the peers to become servers. Therefore, all peers have an economic incentive to be only clients, not servers, as serving files costs resources without bringing benefit. Existing peer-to-peer protocols use a credit system enforcing the peer to serve some content to the community before downloading more.