CS 550: Advanced Operating Systems

Work realized by

Florentin Bekier
Rémi Blaise

Consistent P2P File Sharing System: Design Document

Taught by
Dr. Zhiling Lan

Master of Computer Science, Spring 2020

Contents

1 General architecture

The Consistent P2P File Sharing System is an extension of the previous Gnutella P2P File Sharing System made for the second programming assignment. Therefore, the general architecture and specifications are the same as before. In this document we will detail the changes between the second version and this one.

2 Technology specifications

All the other technologies used are the same that in the previous assignments. In this version, we reintroduced a SQLite database in the **Peer** because we needed to store some information about the files owned and downloaded such as the number of version, and the origin server ID.

3 Configuration

Because our software became more complex over the time, we now have many different keys in our configuration files. Let's review them.

3.1 Super-Peer

- `port`: Port number of the super-peer (introduced in the first version),
- `keyStorageDir`: Directory where to store the public keys of the peer (introducted in the first version),
- `enableSignatureChecks`: Specify if we need to check for the signature (introduced in the first version),
- `logAllDatabase`: Specify if we log all the database in the console (introduced in the first version),
- `queryLifetime`: Time to wait before getting rid of the messages stored in history log (introduced in the second version),
- `leafNodes`: List of the leaf-nodes in the sub-network (introduced in the second version),
- `neighbors`: List of the others super-peers in the network (introduced in the second version),
- `strategy`: Consistency strategy to use. 0 is push-based, 1 is pull-based with leaf-node cache and 2 is pull-based with super-peer cache (new in this version).

3.2 Peer

- `indexHost`: Host name of the super-peer (introduced in the first version),
- `indexPort`: Port number of the super-peer (introduced in the first version),
- `port`: Port number of the peer (introduced in the first version),
- `sharedDir`: Directory where to store the files owned by this peer (introduced in the first version but different behavior in this version),

- `downloadDir`: Directory where to store the files downloaded by the peer and not owned (new in this version),
- `keyStorageDir`: Directory where to store the private key of the peer (introducted in the first version),
- `ttl`: Time-to-live for the search query (introduced in the second version),
- `queryLifetime`: Time to wait to received the results of a search query (introduced in the second version),
- `strategy`: Consistency strategy to use. 0 is push-based, 1 is pull-based with leaf-node cache and 2 is pull-based with super-peer cache (new in this version),
- `ttr`: Time-to-refresh the downloaded files (new in this version).

4 Design of the Super-Peer

The Super-Peer 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 Super-Peer API.
- The `repository.js` file handles the persistence layer by handling the arrays that save the data.
- The `interface.js` file implements the client communication protocol.
- The `search.js` file handles the file search (local search and propagation).
- The `config.js` file reads the configuration.

4.1 API

Using the previously defined protocol, the Super-Peer 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 contact address (IP and port),
- its file list,
- the SHA-256 signature of the request certifying the identity of the peer.

Expected request:

```
{
    "name": "registry",
    "parameters": {
        "ip": "string",
        "port": integer,
        "files": [
            {
                "name": "string"
            },
            ...
        ],
        "signature": "string"
    }
}
```

Expected response:

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

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 Super-Peer, 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 `repository.js` file handles the persistence layer by handling the SQLite database that save the files data.
- The `config.js` file reads and initializes the configuration.

5.1 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. The user can then select the file he wants to download or return to the menu.

3. If in push-based approach, refresh the files.
4. 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 Push-based approach

In the push-based approach, the Peer software will monitored the shared folder for any change in the files and if a change occurs, it will increment the version number of the file and send an `invalidate` request to its Super-Peer that will propagate it through the network. If a Peer receives an `invalidate` request, it will check if it has the file and if so, it will discard it by sending a new `registry` request without the file.

6.1 The `invalidate` procedure

Invalidate a file. This request is sent by the origin server each time an owned file is modified. It can be sent and received by both the Peer and the Super-Peer.

Expected request:

```
{
    "name": "invalidate",
    "parameters": {
        "messageId": "string",
        "fileName": "string",
        "version": integer,
        "ip": "string",
        "port": integer
    }
}
```

Expected response:

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

7 Pull-based approach with leaf-node cache

In the first pull-based approach, the Peer software with regularly poll the origin server (based on a TTR value) to see if the file is still up to date or not. We decided to perform the poll in an eager manner by using timers.
It is also possible to manually refresh the files by choosing the corresponding option in the CLI menu.

7.1 The `poll` procedure

Poll the origin server of a file to see if it is up to date or not.

Expected request:

```
{
    "name": "poll",
    "parameters": {
        "fileName": "string",
        "version": integer
    }
}
```

Expected response:

```
{
    "status": "success",
    "data": {
        "upToDate": boolean,
        "ttr": integer,
        "lastModifiedTime": "string"
    }
}
```

8 Pull-based approach with super-peer cache

In the second pull-based approach, it is the Super-Peer that will poll the other super-peers to check if a new version is available and then notify the relevant leaf-nodes. This can be done with the following steps:

1. When a change is made, the origin server send an `invalidate` request to its Super-Peer that will update the version number of the file.
2. When the TTR of a file is outdated, a Super-Peer will contact the others via a `poll` request to know if the file is still up to date or not.
3. If the Super-Peer receive at least one "out of date" reply, it will contact the leaf-nodes via an `invalidate` request and the leaf-node will discard the file.

We chose this approach because it was the one that reduced the most the number APIs to provide. There is only to adapt the existing `poll` procedure to work between super-peers.

8.1 The `poll` procedure

Poll a Super-Peer to see if it is up to date or not.

Expected request:

```
{
    "name": "poll",
    "parameters": {
        "fileName": "string",
        "version": integer
    }
}
```

Expected response:

```
{
    "status": "success",
    "data": {
        "outOfDate": boolean,
        "notFound": boolean
    }
}
```

9 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 Super-Peer 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 Super-Peer.

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 (statically stored) and cryptographically sign any further call to the `register` procedure.

10 Possible improvements

To go further, the following improvements could be brought to our P2P 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 Super-Peer could sort available server-Peers to better distribute the global charge over all servers.
- The Super-Peer could include a web-browsable interface listing available files on the system.

- The Super-Peer 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.
- The list of neighbors and leaf-nodes of a Super-Peer could be dynamic with a database instead of static in the configuration file.
- Because we added features at different steps for the three assignments, the code has become complicated and could be optimized and simplified.