ILLINOIS INSTITUTE OF TECHNOLOGY

CS 550: Advanced Operating Systems

Work realized by

Florentin Bekier
Rémi Blaise

Programming Assignment 4: Design Document

Taught by
Dr. Zhiling Lan

Master of Computer Science, Spring 2020

Contents

1 General architecture

As in the previous assignments, our project is made of 2 software:

1. The **Super-Peer** centralizes an index of all available files in its leaf nodes. Each running instance of the Super-Peer creates a new independent file-sharing network and can communicate with the other super-peers.

2. The **Peer** can download and serve files from/to the network. Each running instance is a node of the network and is a leaf-node of a Super-Peer.

To simplify the deployment of a network with several super-peers, we created an installer software that can generate as many super-peers and leaf-nodes as we want according to either the all-to-all or linear topology.

This project also implements the push-based approach implemented in Programming Assignment 3 to ensure the consistency of the files. The goal of this specific assignment was to add RSA encryption into our project. This document explains all the design choices since the first assignment.

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 three different modes: **Super-Peer** ↔ **Peer**, **Peer** ↔ **Peer** and **Super-Peer** ↔ **Super-Peer**. In every case, 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 that is encrypted via RSA (see section 4) through a socket. The protocol is designed to enable the server to expose an **API** made of **procedures**.

When a peer wants to communicate with another one (either Peer or Super-Peer), it is considered as a client and the other one is considered as a server. The communication is done through the following steps:

1. The client opens the connection over TCP.
2. The client emits a request containing the cypher of a JSON message (see section 3.1).
3. The server replies with a cypher of a Response JSON Message (see section 3.2).
4. The server closes the connection.

3.1 Request message format

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

3.2 Response message format

Success message:

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

Error message:

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

4 RSA encryption

This section will detail the implementation of the RSA encryption algorithm made for this assignment. The module that we created is composed of 3 submodules:

1. `asn1.js` is used to decode the Distinguished Encoding Rules (DER) encoding of ASN.1 used to represent keys in PEM format.
2. `rsa.js` contains the core functions of the algorithm: key generation and import, encryption and decryption.
3. `rsa-keypair.js` contains functions to write/read the key files on disk.

4.1 Key generation and import

To generate the RSA keypairs (public and private key) we used the keypair module that allows us to obtain a RSA PEM key pair of a specific length. The default length in the project is 1024 bits but it can easily be changed to 2048 for example. Once we generate the key pair, we can save it on disk in `.pem` files and import it in order to extract the public key modulus, public key exponent and private key exponent. This is done via the `importKey` function that takes a public or private key PEM string and return a corresponding key object by decoding it using the `asn1.js` module. The only supported format is PKCS#1.

4.2 Encryption and decryption

The basic RSA encryption and decryption is done via the internal `encrypt` and `decrypt` functions. However, those functions only allow to encrypt a number into a cypher that is also a number. Therefore, the public functions are `encryptText` and `decryptText` that accept the text message or cypher and a key object and also return a string. The steps for encoding a message are the following (decoding steps are identical but reversed):

1. Break the message into chunks of maximum size (not recommended but necessary to encode messages of variable length)
2. Perform EME-OAEP encoding method (based on Optimal Asymmetric Encryption Padding scheme) for better security
3. Convert the encoded message to an integer message representative
4. Apply RSA encryption on the integer message representative to produce an integer ciphertext representative
5. Convert the ciphertext representative to a ciphertext of length k octets (length of the key)

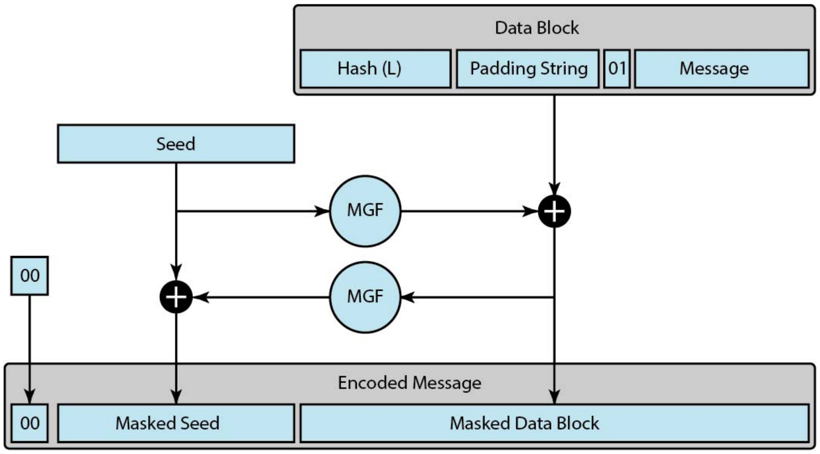

Figure 1: EME-OAEP encoding method (Wikimedia Commons)

Our EME-OAEP implementation uses SHA-256 as hash function and MGF1 as mask generation function. We will not go over EME-OAEP in detail here but the figure above summarizes how a message is encoded. We chose to use this encoding method because it is one of the most secure for RSA. However, it is not perfect either, especially because we are not using hybrid encryption.

5 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 `config.js` file reads the configuration.
- The `interface.js` file implements the communication protocol.
- The `keystore.js` file implements a lightweight store for public keys.
- The `procedures.js` file defines the actual procedures provided by the Index API.
- The `repository.js` file handles the persistence layer of the leaf-nodes' files and message history via in-memory arrays.
- The `search.js` file implements some utility function for the propagation of the `search`, `queryhit` and `invalidate` procedures.

5.1 API

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

5.1.1 `pks`

To communicate via RSA, the peers must exchange their public key via the PKS (public key sharing) procedure. It is used in only two cases:

1. At the startup of a leaf-node to share its public key with its super-peer and receive the public key of the super-peer in return
2. When a super-peer need to contact another super-peer for the first time

It is the only procedure that is unencrypted. The response, however, is encrypted.

Expected request:

```
{
    "name": "pks",
    "parameters": {
        "id": "{ip}:{peer}",
        "key": "string"
    }
}
```

Expected response:

```
{
    "status": "success",
    "data": {
        "id": "{ip}:{peer}",
        "key": "string"
    }
}
```

5.1.2 registry

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) and its file list.

Expected request:

```
{
    "name": "registry",
    "parameters": {
        "id": "{ip}:{peer}",
        "files": [
            {
                "name": "string",
                "version": integer
            },
            ...
        ]
    }
}
```

Expected response:

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

5.1.3 search

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": {
        "id": "{ip}:{peer}",
        "messageId": "string",
        "ttl": integer,
        "fileName": "string"
    }
}
```

Expected response:

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

5.1.4 queryhit

If a Super-Peer receives a `queryhit`, it will backpropagate the request to the sender of the original `search` request. To find the sender information, it will use the message ID to look into the message history and find the original message data. Note that, this time, `ip` and `port` correspond to the result to backpropagate and not to the sender information.

Expected request:

```
{
    "name": "queryhit",
    "parameters": {
        "id": "{ip}:{peer}",
        "messageId": "string",
        "fileName": "string",
        "ip": "string",
        "port": integer,
        "key:" "string"
    }
}
```

Expected response:

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

5.1.5 invalidate

Invalidate a file. This request is sent by the origin server each time an owned file is modified. When a super-peer receives this request, it will log the message like for the `queryhit` procedure and propagate it to all its neighbors and leaf-node if it is the first time it receives it.

Expected request:

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

Expected response:

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

5.2 Configuration

Here is the list of configuration keys with their explanation:

- `port`: Port number of the super-peer (introduced in the first version),
- `keyStorageDir`: Directory where to store the public keys of the peer (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),
- `debugRSA`: Enable or disable the RSA logs in the console (new in this version).

6 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 `config.js` file reads and initializes the configuration.
- The `files.js` file handles the file sharing management.
- The `interface.js` file implements the communication protocol.

- The `keystore.js` file implements a lightweight store for public keys.
- The `repository.js` file handles the persistence layer to save the file information by communicating with the database through the Sequelize ORM.
- The `server.js` file implements the server side: it instanciates the TCP server and listens for file requests.

6.1 API

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

6.1.1 `retrieve`

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

Expected request:

```
{
    "name": "retrieve",
    "parameters": {
        "id": "{ip}:{peer}",
        "fileName": "string"
    }
}
```

Expected response:

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

6.1.2 `queryhit`

After performing a `search` request to its Super-Peer, the Peer will listen for incoming `queryhit` requests that will deliver the result. The listening time is specified in the configuration file.

The expected request and response are identical to the Super-Peer's API, see section 5.1.4.

6.1.3 `invalidate`

When a Peer receives an `invalidate` request, it will discard the corresponding file or ignore it if it doesn't have the file.

The expected request and response are identical to the Super-Peer's API, see section 5.1.5.

6.2 Configuration

Here is the list of configuration keys with their explanation:

- `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 (introduced in the third version),
- `keyStorageDir`: Directory where to store the private key of the peer (introduced 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),
- `debugRSA`: Enable or disable the RSA logs in the console (new in this version).

6.3 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.

7 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 four assignments, the code has become complicated and could be optimized and simplified.
- The RSA implementation is not perfect because it is recommended to use a hybrid encryption with RSA (only encrypting a symmetric key in RSA) for better performances and security. Moreover, creating its own cryptography library is not recommended in a real-life environment because of all the flaws it may have.