2 Implementation

2.1 Filesystem Structure

For the filesystem structure we adopted the structure of the well known version control system `git`. We save all the persistent data in the users directory in `.por`. The prover and the verifier have different subdirectories though, namely `objects` for the prover and `files` for the verifier. For the verifier and the prover store all their configurations in `por.conf`. The figure 2.1 describes this in more detail.

The verifier saves a file for every upload he makes in the `files` directory with the 64-digits hexadecimal root-hash as its filename. This file is a key-value-store for the values `filesize` in bytes, `challenges` for the challenges object serialized and BASE64 coded, `blockCount`, the `keys` object serialized and BASE64 coded, `hash` of the tree root, `name` of the original file, `blockSize` and `depth` of the merkle-tree.

The prover saves all data (blocks and merkle-trees) in the `object` directory. The first two digits of the hexadecimal resentation of the block-hash resp. the root-hash define the subdirectory, all remaining digits define the actual filename.

```
.por
├── por.conf
├── files
│   └── <64-digit root-hash as filename>
└── objects
    ├── ab
    │   └── <62-digit root-hash for merkle-tree file>
    └── c4
        └── <126-digit block-hash for block>
```

Fig. 2.1: filesystem structure.

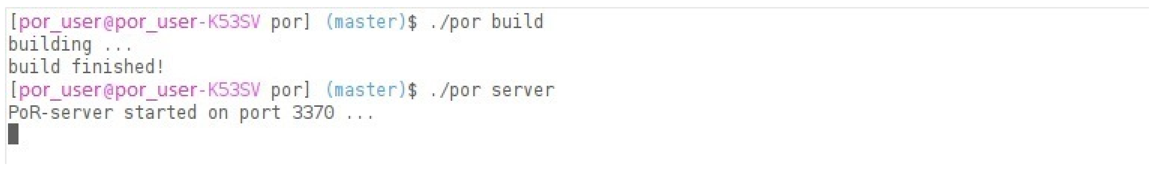
2.2 User Interface

We wanted the userinterface to be small and simple. So we decided to make a console-tool, because a graphical-user-interface would have been an overhead and is not necessary for the programs purpose. Furthermore a console-tool like `git` is very easy to use. As mentioned we have only a few commands necessary to use our application. Every command starts with "./por" which stands for "prove of retrievability". Before we can get started the application has to be compiled. Therefore we use the build-management-tool Maven as we chose Java to be our implementation-language. To build the application we open our console, change into the corresponding directory and type "./por build" (see figure 2.2). This command will cause a small shell-script to compile the source-code for us and create the servers storage-directory with the name ".por". If the build was successful we are able to use the following set of commands:

- `server`: starts the server (prover).
- `create <filepath>`: prepares and sends a new file to the prover (encode).
- `verify <file-hash> <amount of challenges>`: challenges the prover for a file.
- `update <file-hash> <block-to-update> <file>`: sends an update for to the prover.
- `download <file-hash> <file-path>`: extracts the whole file from the prover.
- `delete <file-hash>`: deletes a file at the prover.
- `list`: shows a list of files stored at the prover.
- `config`: shows configurable parameters.
- `config <key> [<value>]`: configures a certain parameter.
- `usage`: shows a list of available commands.

2.2.1 Usage

First we have to initialize the server (the prover) by typing "./por server" (see figure 2.2). This will create a server-socket on a configurable port and a corresponding thread which will handle all our requests to it.



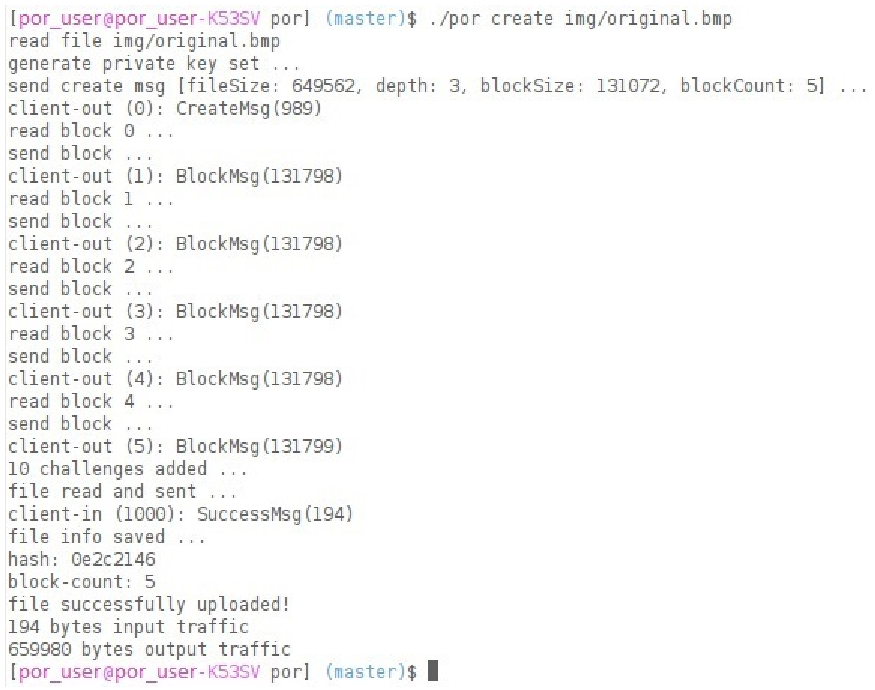
```
[por_user@por_user-K53SV por] (master)$ ./por build
building ...
build finished!
[por_user@por_user-K53SV por] (master)$ ./por server
PoR-server started on port 3370 ...
```

Fig. 2.2: Building por-application and starting server.

Now we can upload new files to the prover by using "./por create <filepath>". This will create a random amount of challenges for our file before we send it blockwise to the prover. Therefore the server-socket receives an initial message (`CreateMsg`) and then our file-blocks within several messages of type `BlockMsg`. The first message contains all the meta-data about our file necessary for the prover and the following contain the data. The amount of blocks can be determined in the configuration. If the file has been stored successfully at the prover, we receive a success-message (`SuccessMsg`). This message contains informations like the file-hash and amount of blocks that have been stored. With this information we can already determine if the file has been stored consistent at the prover (see figure 2.3).

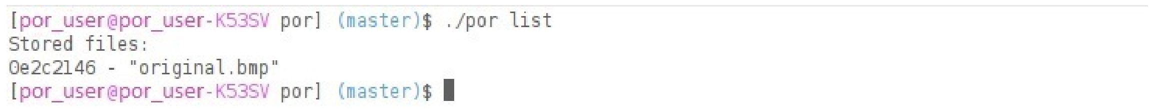
With the command "./por list" we receive a list of all uploaded files and their hashes (see figure 2.4). The filehash is the identifier of our file at the prover and thus important for further actions.

Now we can check if our file is still consistent with the command "./por verify <filehash> <amount of challenges>" (see figure 2.5). With the parameter "<filehash>" we determine which file we want to prove consistent and "<amount of challenges>" lets us define how many challenges we want to send to the prover. As mentioned before we have only a random amount of



```
[por_user@por_user-K53SV por] (master)$ ./por create img/original.bmp
read file img/original.bmp
generate private key set ...
send create msg [fileSize: 649562, depth: 3, blockSize: 131072, blockCount: 5] ...
client-out (0): CreateMsg(989)
read block 0 ...
send block ...
client-out (1): BlockMsg(131798)
read block 1 ...
send block ...
client-out (2): BlockMsg(131798)
read block 2 ...
send block ...
client-out (3): BlockMsg(131798)
read block 3 ...
send block ...
client-out (4): BlockMsg(131798)
read block 4 ...
send block ...
client-out (5): BlockMsg(131799)
10 challenges added ...
file read and sent ...
client-in (1000): SuccessMsg(194)
file info saved ...
hash: 0e2c2146
block-count: 5
file successfully uploaded!
194 bytes input traffic
659980 bytes output traffic
[por_user@por_user-K53SV por] (master)$
```

Fig. 2.3: Uploading a new file to the server.



```
[por_user@por_user-K53SV por] (master)$ ./por list
Stored files:
0e2c2146 - "original.bmp"
[por_user@por_user-K53SV por] (master)$
```

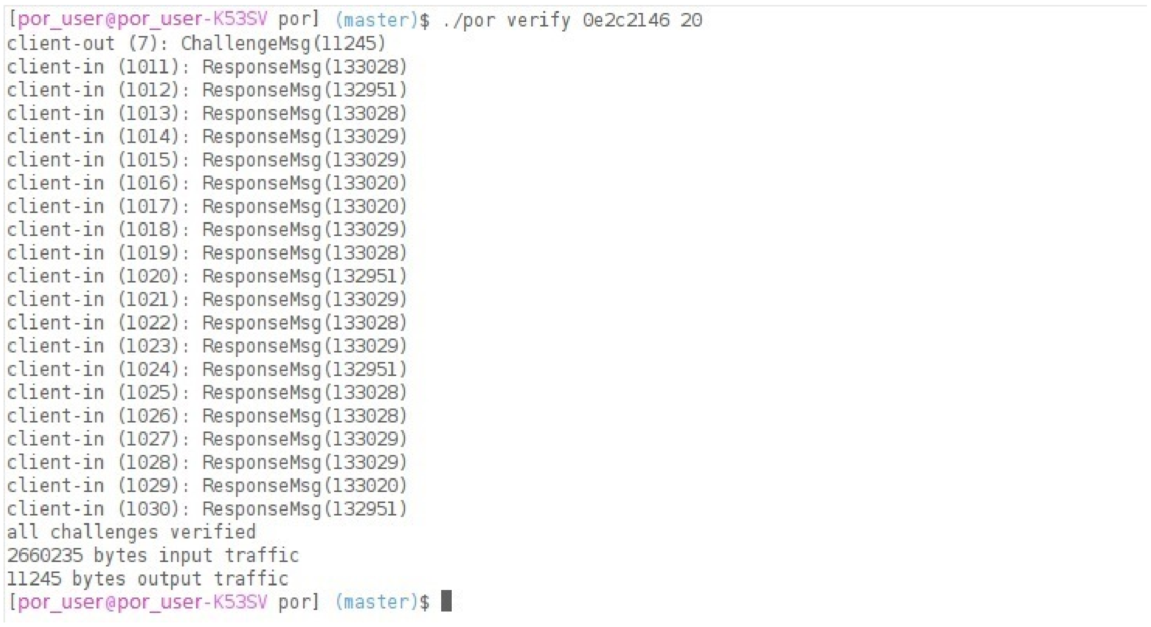
Fig. 2.4: A list of uploaded files.

challenges for a file. If we want to send a challenge for a block where we have none left or never had one, the application starts to send pseudo-challenges which are taken as a real challenge by the prover. Then we receive response-messages (`ResponseMsg`) which include the answers to our sent challenges and check them. If the file is still consistent, all responses will pass the test.

To simulate a corrupted prover we can manipulate a fileblock in the provers directory ".por objects". To this end, we choose one of the file blocks in one of the folders but not the one named after the file hash like described in section "File Structure". This folder contains the serialized file information for the prover. If we verify a file now, the prover fails in responding correctly (see figure 2.6). In order to get the manipulated block, we might have to verify multiple times or with a higher challenge amount.

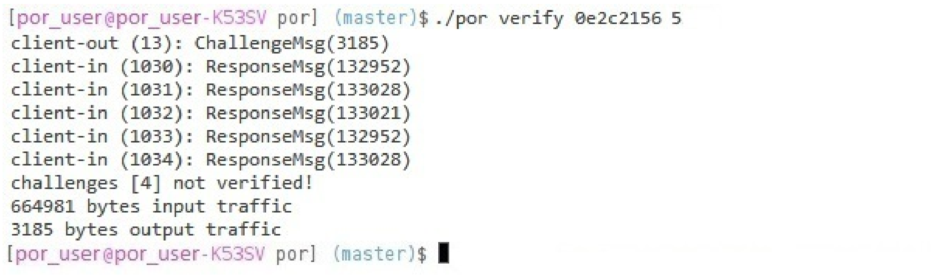
If we want to change a file we can do so with the command "./por update <file-hash> <block-to-update> <file>" (see figure 2.7). The first parameter again is for the prover to identify the right file. Our file is stored in blocks at the prover so with the second parameter we define which block of our file should be changed. The third and last parameter should be the path of our updated file we want to upload. Then the determined block of the updated file will be sent to the prover where it replaces the corresponding block of the original file.

To get our file back we simply use the command "./por download <filehash>" (see figure 2.8). This will create a challenge-message for each file-block of our file and send it to the prover. The



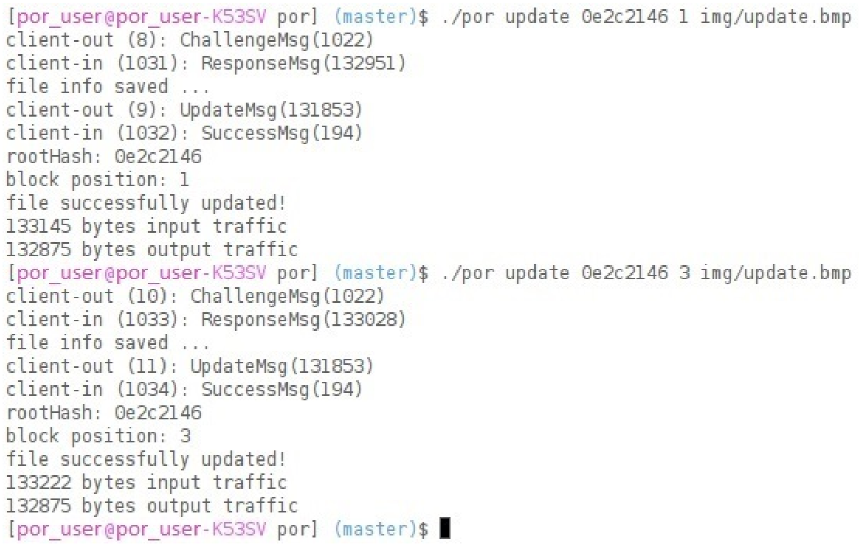
```
[por_user@por_user-K53SV por] (master)$ ./por verify 0e2c2146 20
client-out (7): ChallengeMsg(11245)
client-in (1011): ResponseMsg(133028)
client-in (1012): ResponseMsg(132951)
client-in (1013): ResponseMsg(133028)
client-in (1014): ResponseMsg(133029)
client-in (1015): ResponseMsg(133029)
client-in (1016): ResponseMsg(133020)
client-in (1017): ResponseMsg(133020)
client-in (1018): ResponseMsg(133029)
client-in (1019): ResponseMsg(133028)
client-in (1020): ResponseMsg(132951)
client-in (1021): ResponseMsg(133029)
client-in (1022): ResponseMsg(133028)
client-in (1023): ResponseMsg(133029)
client-in (1024): ResponseMsg(132951)
client-in (1025): ResponseMsg(133028)
client-in (1026): ResponseMsg(133028)
client-in (1027): ResponseMsg(133029)
client-in (1028): ResponseMsg(133029)
client-in (1029): ResponseMsg(133020)
client-in (1030): ResponseMsg(132951)
all challenges verified
2660235 bytes input traffic
11245 bytes output traffic
[por_user@por_user-K53SV por] (master)$
```

Fig. 2.5: Verifing an uploaded file with 20 challenges.



```
[por_user@por_user-K53SV por] (master)$ ./por verify 0e2c2156 5
client-out (13): ChallengeMsg(3185)
client-in (1030): ResponseMsg(132952)
client-in (1031): ResponseMsg(133028)
client-in (1032): ResponseMsg(133021)
client-in (1033): ResponseMsg(132952)
client-in (1034): ResponseMsg(133028)
challenges [4] not verified!
664981 bytes input traffic
3185 bytes output traffic
[por_user@por_user-K53SV por] (master)$
```

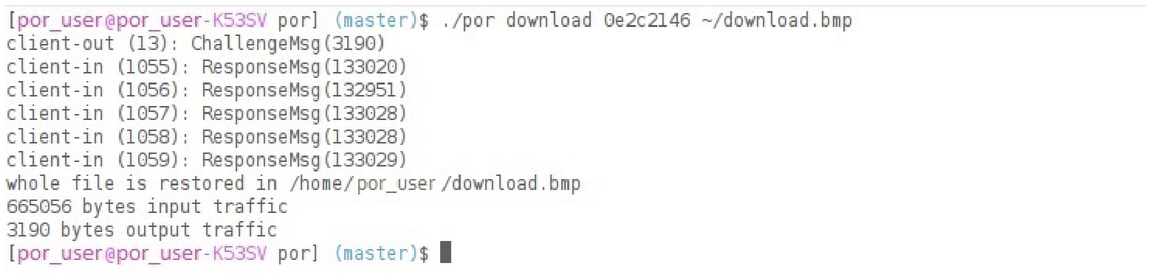
Fig. 2.6: Verifing a manipulated file.



```
[por_user@por_user-K53SV por] (master)$ ./por update 0e2c2146 1 img/update.bmp
client-out (8): ChallengeMsg(1022)
client-in (1031): ResponseMsg(132951)
file info saved ...
client-out (9): UpdateMsg(131853)
client-in (1032): SuccessMsg(194)
rootHash: 0e2c2146
block position: 1
file successfully updated!
133145 bytes input traffic
132875 bytes output traffic
[por_user@por_user-K53SV por] (master)$ ./por update 0e2c2146 3 img/update.bmp
client-out (10): ChallengeMsg(1022)
client-in (1033): ResponseMsg(133028)
file info saved ...
client-out (11): UpdateMsg(131853)
client-in (1034): SuccessMsg(194)
rootHash: 0e2c2146
block position: 3
file successfully updated!
133222 bytes input traffic
132875 bytes output traffic
[por_user@por_user-K53SV por] (master)$
```

Fig. 2.7: Updating the blocks 1 and 3 of the original file with another file.

response-messages we receive do not only contain the answer for the challenges but also the corresponding file-block itself. This means we can restore the whole file out of these messages.



```
[por_user@por_user-K53SV por] (master)$ ./por download 0e2c2146 ~/download.bmp
client-out (13): ChallengeMsg(3190)
client-in (1055): ResponseMsg(133020)
client-in (1056): ResponseMsg(132951)
client-in (1057): ResponseMsg(133028)
client-in (1058): ResponseMsg(133028)
client-in (1059): ResponseMsg(133029)
whole file is restored in /home/por_user/download.bmp
665056 bytes input traffic
3190 bytes output traffic
[por_user@por_user-K53SV por] (master)$
```

Fig. 2.8: Downloading the file "original.bmp".

Here an the example-file of our whole upoload-update-download-scenario:

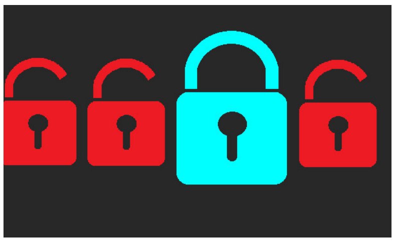


Fig. 2.9: original.bmp

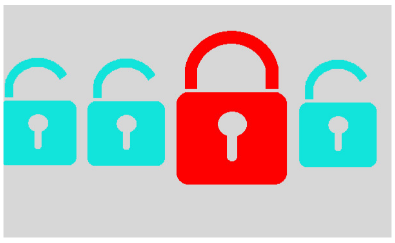


Fig. 2.10: update.bmp

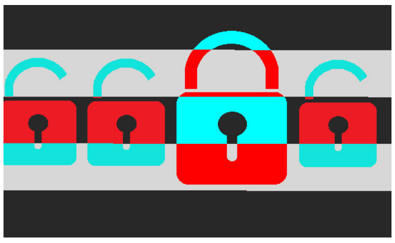
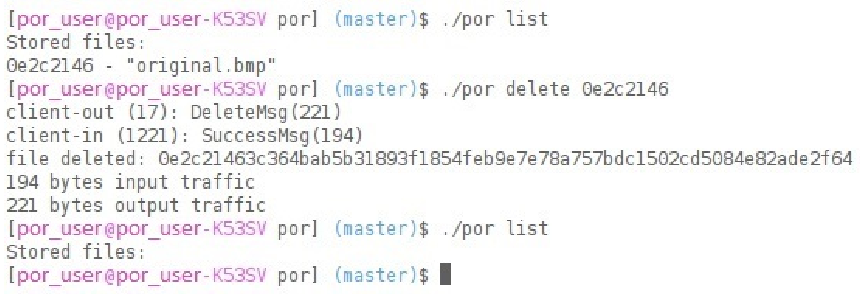


Fig. 2.11: download.bmp

As the "download.bmp" (see figure 2.11) shows, the image has been separated in 5 data-blocks with indices $0, \ldots, 4$ and we updated block no. 1 and 3.

We can also delete files at the prover with "./por delete <file-hash>".



```
[por_user@por_user-K53SV por] (master)$ ./por list
Stored files:
0e2c2146 - "original.bmp"
[por_user@por_user-K53SV por] (master)$ ./por delete 0e2c2146
client-out (17): DeleteMsg(221)
client-in (1221): SuccessMsg(194)
file deleted: 0e2c21463c364bab5b31893f1854feb9e7e78a757bdc1502cd5084e82ade2f64
194 bytes input traffic
221 bytes output traffic
[por_user@por_user-K53SV por] (master)$ ./por list
Stored files:
[por_user@por_user-K53SV por] (master)$
```

Fig. 2.12: Deleting the file "original.bmp".

If one forgets how the syntax of these commands are determined we have prepared a usage-description. By typing "./por usage", "./por" or "./por + randomstring" our application will print a list of possible commands and a short description of them.

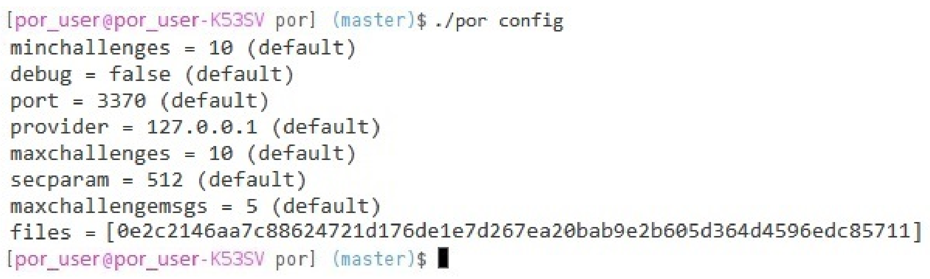
2.2.2 Configuration

The configuration of the application is very simple as it holds only a few parameters to adjust:

- `minchallenges = 10 (default)`: minimum amount of challenges created.
- `debug = false (default)`: debug mode switch that provides more information on the console when using POR commands.
- `port = 3370 (default)`: TCP-port of the provider (prover).

- `provider = 127.0.0.1 (default)`: IP of provider (prover).
- `maxchallenges = 10 (default)`: maximum amount of challenges created.
- `secparam = 512 (default)`: security parameter for initialization of POR. It defines the length of the keys which is in this case 512 bit.
- `maxchallengemsgs = 5 (default)`: maximum challenge messages send to the prover (not used yet, meant for calling verify without parameter ”challenge amount”).
- `files = []`: this parameter holds only the full-length-file-hashes of the uploaded files. It isn't configurable.

By typing ”./por config” (see figure 2.13) one gets a list of these parameters and their current values:



```
[por_user@por_user-K53SV por] (master)$ ./por config
minchallenges = 10 (default)
debug = false (default)
port = 3370 (default)
provider = 127.0.0.1 (default)
maxchallenges = 10 (default)
secparam = 512 (default)
maxchallengemsgs = 5 (default)
files = [0e2c2146aa7c88624721d176de1e7d267ea20bab9e2b605d364d4596edc85711]
[por_user@por_user-K53SV por] (master)$ █
```

Fig. 2.13: Configuration

The amount of challenges that are generated for a file is determined randomly. With the parameters ”minchallenges” and ”maxchallenges” one can define how many challenges will be produced at least or at most. Furthermore it is possible to set the port and the IP-address of the provider. For developing the application one can activate the debug-mode by setting ”debug” to ”true”. This provides more information on the console when using POR commands. For example when `./por verify` is used it does not only show the input and output of messages but also which steps are made in between like fetching a challenge (pseudo or real) or if the response matches the expected hash value. The same goes for `create` and `update`. These provide you with information about new created challenges or forged pre-images. To configure one of these parameters one just types ”./por config <param-name> <param-value>”. So for example if we want to change the value of the security parameter we simply type ”./por config secparam 1024”.

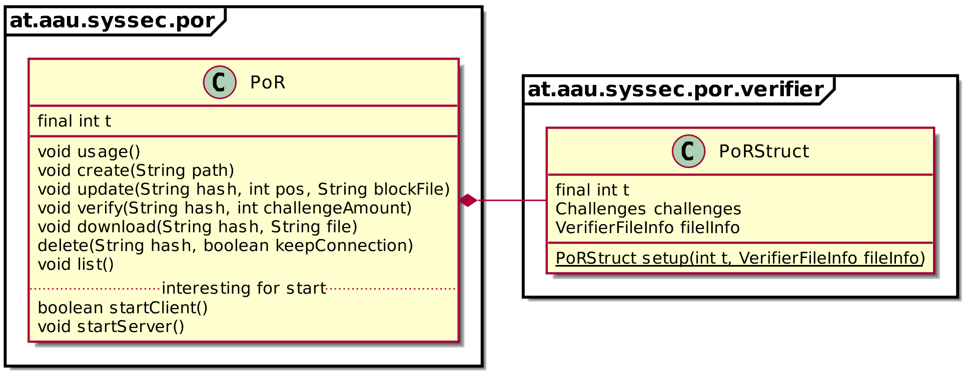
2.3 Classdiagramm and Program Architecture

First we decided to implement the whole application in C++ because of performance improvements. But then it had to be interoperable on different platforms as well. So we found Java more suitable for our purpose. There are supporting libraries such as Bouncy Castle, Java Cryptography Extension, Merkle Tree API and SHA3. The architecture of our implementation consists of the following main classes:

- `PoR`: simulates the whole por-scenario.
- `PoRStruct`: includes all the actions that the verifier can perform.

- `AbstractHandler`: this class is extended by many handlertypes which handle incoming messages from the prover.
- `ClientHandler`: represents the prover and handles all the incoming requests from the verifier.
- `MerkleTree`: is a structure that both verifier and prover use for calculating the hashvalue of a file.
- `ChameleonHash`: this class is responsible for the hashing-calculations.

The `PoR` class, shown in figure 2.14 initiates the whole application by loading the default configuration and defining the secure parameter t.



at.aau.syssec.por
C PoR
final int t
void usage()
void create(String path)
void update(String hash, int pos, String blockFile)
void verify(String hash, int challengeAmount)
void download(String hash, String file)
delete(String hash, boolean keepConnection)
void list()
interesting for start
boolean startClient()
void startServer()
at.aau.syssec.por.verifier
C PoRStruct
final int t
Challenges challenges
VerifierFileInfo fileInfo
PoRStruct setup(int t, VerifierFileInfo fileInfo)

Fig. 2.14: Class `PoR` and `PoRStruct` (drawn with Intellij IDEA plugin "Plant UML").

Furthermore, a client-socket gets started and connects to the serversocket if already active. By client we mean the role of the verifier and with server we mean the role of the prover. The server gets started by command which includes the server-socket for the communication and a handler-thread that handles all the incoming requests sent by the client. The data- and message-exchange between client(verifier) and server(prover) occur through socket-communication. Because of the different requests and responses the verifier and the prover can send, we defined the following message types, all of which extend the class `Message`:

- `BlockMsg`: contains a data-block of a file (can be received from both).
- `ChallengeMsg`: contains a challenge for the prover.
- `ResponseMsg`: contains a response which is returned by the prover for a given challenge.
- `CreateMsg`: is meant as an upload-request at the prover.
- `DeleteMsg`: is a delete-request at the prover.
- `SuccessMsg`: this is a feedback message for the verifier if an action was executed correctly.
- `ErrorMsg`: this is a feedback message for the verifier if an action has failed.

Once the server is started, the client can execute several actions provided by the `PoRStruct`-class which includes:

- `setup`: gets automatically called when the application starts.
- `encode`: creates challenges and sends a file blockwise the prover.

- `challenge`: fetches a challenge out of the verifier's set.
- `verify`: sends a certain amount of challenges to the prover and checks the responses.
- `update`: updates a file block at the prover.
- `extract`: downloads a file from the prover.

The method `setup` sets the security parameter t which is necessary for initiating the `PoRStruct`. The function `encode` is responsible for uploading a new file to the server. To do so, the file gets split up into file-blocks. The size of these blocks is determined by the class `BlockSize` which gets it's parameters from the class `Config`, which persists its information in an object `KeyValueFile`. Further the class `BlockSize` calculates the amount of blocks that will be created and thus the depth of the structure `MerkleTree` can be determined. Then an instance of the tree is created and filled with the file-blocks. Once the tree is finished, some challenge-response-pairs are generated randomly out of the file-blocks within the tree-structure. Randomly means that there may exist one or more challenges for some of the blocks, not necessarily for all of them. These challenge-response-pairs are stored in the `VerifierFileInfo`-class shown in figure 2.15.

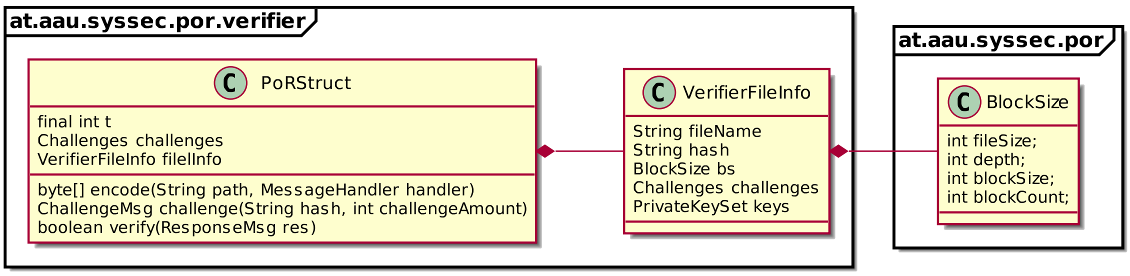


Fig. 2.15: Class `VerifierFileInfo`

This class contains all the information about a file. This includes the filename, the root-hash of it's Merkle-tree-structure, information about the `BlockSize`, a list of challenge-response-pairs `Challenges` and an instance of `PrivateKeySet` which will be explained further below.

Then the server gets an information-message (`CreateMsg`) about how many blocks the file consists of, and the file gets transmitted to the server blockwise (`BlockMsg`).

Now the file can be remote verified as being consistent with the method `challenge` and `verify`. The first method picks randomly one or more of the generated challenge-response-pairs so we can send them to the server via a `ChallengeMsg`. If there isn't any challenge-response-pair stored for a block, this method generates a pseudo-challenge. A pseudo-challenge in our case is a challenge we don't know the response for it but the prover doesn't know that. Listing 2.16 shows how such a pseudo-challenge is created.

So the function checks if there are any challenges available for a position i of the data. If a non used challenge is found it will be marked used and returned. If none is found a pseudo challenge is added for this position.

The `verify`-method checks wether the response of the server to its corresponding challenge is correct or not. If the response was created out of a pseudo-challenge, we assume that it is

```java
/**
* Returns a challenge for a certain block-position. If no challenge is
* available for this position, a pseudo-challenge is returned.
*
* @param pos - block-position
* @return Challenge
*/
public Challenge getChallenge(int pos) {
    List<VerifierChallenge> list = challenges.get(pos);

    if (list != null && list.size() > 0) {
        for (VerifierChallenge challenge : list) {
            if (!challenge.isSent()) {
                challenge.send();
                return new Challenge(pos, challenge.getChallenge());
            }
        }
    }

    byte[] challenge = Util.generateChallenge(Config
    .getIntValue(Config.SEC_PARAM));
    Challenge pseudoChallenge = new Challenge(pos, challenge);
    addPseudoChallenge(pos, pseudoChallenge);
    return pseudoChallenge;
}
```

Fig. 2.16: Checks for challenges for a certain position, adds pseudo challenges if none exist.

correct as long as it belongs to the same block as the challenge.

The function `update` speaks for itself as it prepares an update which is to be send to the server. It contains all the information for the prover to perform this update. Additionally and more important this method deletes all challenge-response-pairs for the updated block as they become useless and generates new ones during the update-process at the client's side. For simplicity an update allows only for change of a file in our implementation. We didn't cover file extensions in this work.

Last but not least the `extract`-method prepares a set of challenges, one challenge for each block of a file, in order to download the whole file. The download occurs through our challenge-response-procedure as the response contains not only the response itself, but also the corresponding file-block and thus we can restore the whole file out of the responses. Additionally we check for integrity of the file by comparing its root hash to the original stored one.

The `AbstractHandler` - class (see fig. 2.17) is meant for handling the different types of incoming messages at the verifier. Therefore it gets extended by the following classes:

- `ChallengeHandler`: This handler gets active after the verifier has sent one or more challenges to the prover. It expects one or more response messages, corresponding to the challenges sent and checks if all response-values are equal to the expected ones.
- `EncodeHandler`: This handler is waiting for a success-message in order to check wether the upload of the file has been successfully or not. Since the verifier does not trust the prover in anyway, this success-message includes the file-hash of the uploaded file so that we can check if the prover has persisted our file correctly. Only if the server passes this test, the encode-handler persists the fileinfo at the client.
- `DeleteHandler`: Similar as the encode-handler, the delete-handler waits for a success-message if our file has been deleted at the server.
- `UpdateHandler`: This handler waits for a response-message after a challenge has been sent because before we want to update a file-block we would like to generate some challenges for the new file-block as the current challenges for this block become useless after the update. In order to do so, we need some information of that block which is contained in such a response-message.
- `DownloadHandler`: This handler waits for n responses where the variable stands for the amount of blocks a file has. After the verifier has sent exactly one challenge for each block of the file, the whole file can be restored out of the n responses as a response-message also contains the file-block itself.

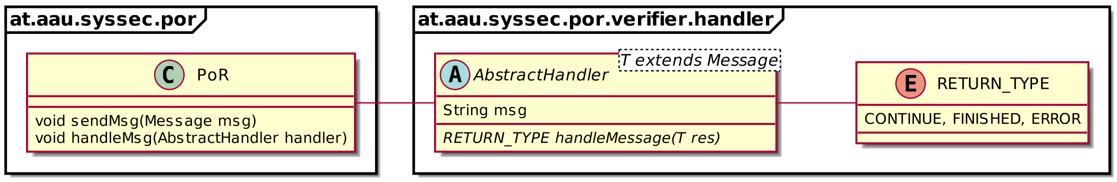

at.aau.syssec.por
C PoR
void sendMsg(Message msg)
void handleMsg(AbstractHandler handler)
at.aau.syssec.por.verifier.handler
A AbstractHandler
T extends Message
String msg
RETURN_TYPE handleMessage(T res)
E RETURN_TYPE
CONTINUE, FINISHED, ERROR

Fig. 2.17: Class `AbstractHandler`

The `Clienthandler` (see figure 2.18) on the other hand is responsible for the prover handling the incoming requests of the verifier. It does not, like the other handling classes, extend `Abstracthandler` but inherits from `Thread`, as it has always to be active. It gets initialized when the server is started.

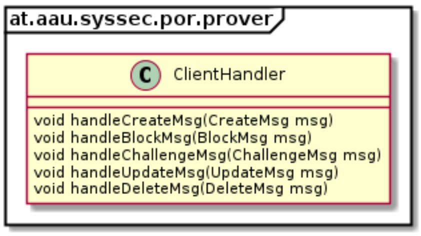

at.aau.syssec.por.prover
C ClientHandler
void handleCreateMsg(CreateMsg msg)
void handleBlockMsg(BlockMsg msg)
void handleChallengeMsg(ChallengeMsg msg)
void handleUpdateMsg(UpdateMsg msg)
void handleDeleteMsg(DeleteMsg msg)

Fig. 2.18: Class `ClientHandler`

At the prover side the file gets stored in form of the same structure as it was built at the verifier, the `MerkleTree` (see figure 2.19). There the file's data-blocks (class: `Block`) are located in the leafs (class: `Leaf`) of the tree. This structure is part of our spot-checking-mechanism and besides it helps us to navigate to single data-spots more efficiently. The tree is a binary one and it's build recursively. To do so we defined an interface `Node` which both, the class `MerkleTree` and the class `Leaf` extend. A `Node` in the tree is either a `Leaf` or a `MerkleTree`, but only the latter one has two children. Both types of nodes have to implement the method `getHash` which returns it's hash value. A `Leaf`-node returns the hash of it's data-block while a `MerkleTree`-node returns the SHA3 encoded concatenation of the hash-values of it's children. So `getHash` called from the root-element of the tree recursively calls this method from it's children until it reaches the leafs.

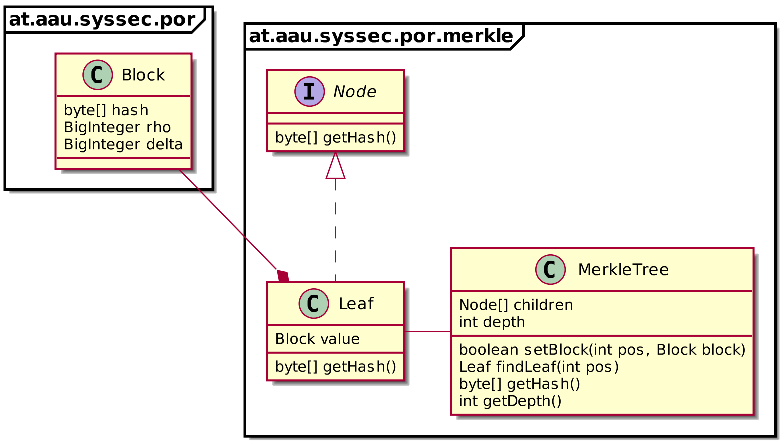


Fig. 2.19: Class `MerkleTree`

First we have to create this tree at the verifier as we want to create our challenges for a certain file out of it. A challenge is created by generating a random `byte[]`. This random array gets concatenated with the data of the file-block then encoded with SHA3 and finally the root-hash of the whole Merkle-Tree is calculated with this new hash. This temporary calculated root-hash which is also the expected response for this challenge, the challenge itself and the position of the corresponding file-block are stored at the verifier to compare the values later with the prover's response.
The tree itself is established depending on the amount of blocks our file gets split into. This information can be retrieved out of an object of type `BlockSize` for the corresponding file. The amount of leafs of our tree is chosen to be the smallest power of two that is greater or equal the number of file-blocks. This implies that the tree might have empty nodes if the amount is greater than block count. In this case if a leaf node is empty it returns an empty byte array instead of a hash value. No empty data block `Block` is stored physically for this node.
Once the tree for a file is established at the prover's side, we can check if it's consistent by asking for the root-hash. If proven consistent, the file and the tree can be deleted at the verifiers side after we created challenges for some of the file-blocks. As mentioned we need the tree to create the challenges but how can we create new ones if the tree exists only at the prover's side? Therefore we implemented a structure within the `MerkleTree` that helps us to gain the necessary information for creating challenges, the so called `Authpath` (see figure 2.20). Everytime we check a spot in the `MerkleTree` by challenging the prover, we get this authentication-path besides the response and the file-block itself in return. This authentication-path contains all the nodes along

the path from the checked leaf to the root-node and their hashes and that's all we need to create new challenges for this leaf-node (see figure 1.4).

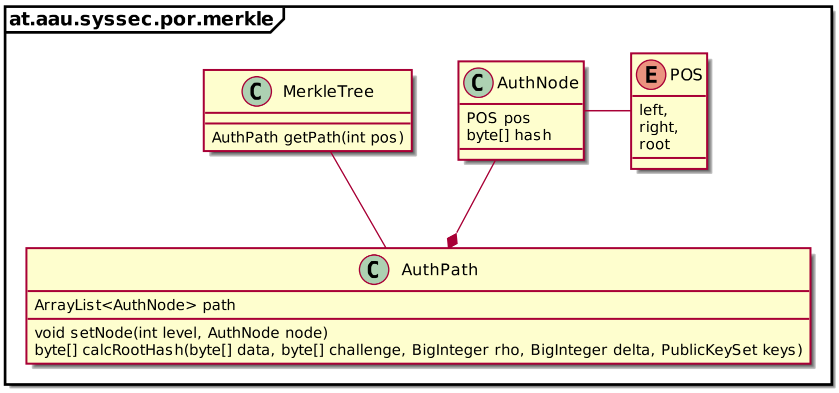

at.aau.syssec.por.merkle
C MerkleTree
AuthPath getPath(int pos)
C AuthNode
POS pos
byte[] hash
E POS
left,
right,
root
C AuthPath
ArrayList<AuthNode> path
void setNode(int level, AuthNode node)
byte[] calcRootHash(byte[] data, byte[] challenge, BigInteger rho, BigInteger delta, PublicKeySet keys)

Fig. 2.20: Class `Authpath`

The class `ChameleonHash` (see figure 2.21) provides all the cryptographic algorithms needed to make this POR dynamic. The function `keyGen(int t)` takes the security parameter t and returns an object `PrivateKeySet` which is meant for the verifier only. It contains all necessary parameters to perform the functions `hash()` and `forge()`. This includes the two primes p and q and the generator element g as well as public and secret key (sk, pk) The verifier extracts all information out of this set except the secret key and sends it in an object `PublicKeySet` to the prover on initialization. This set contains the same as the private key set except the secret key sk. Further details to chameleon hash implementations are found in the section 2.4 "Algorithms".

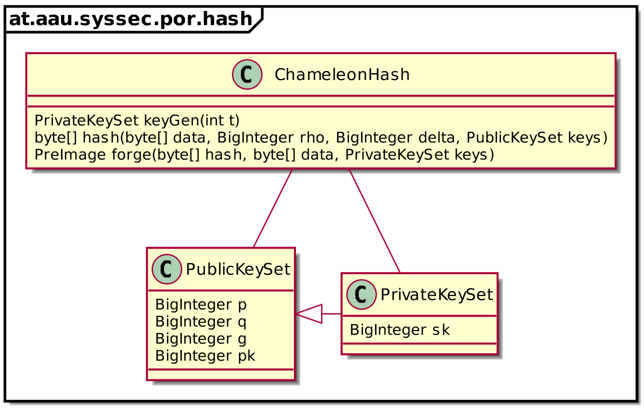

at.aau.syssec.por.hash
C ChameleonHash
PrivateKeySet keyGen(int t)
byte[] hash(byte[] data, BigInteger rho, BigInteger delta, PublicKeySet keys)
PreImage forge(byte[] hash, byte[] data, PrivateKeySet keys)
C PublicKeySet
BigInteger p
BigInteger q
BigInteger g
BigInteger pk
C PrivateKeySet
BigInteger sk

Fig. 2.21: Class `ChameleonHash`, `PublicKeySet` and `PrivateKeySet`.