CS-438 - Decentralized Systems Engineering, Fall 2023

Homework 2

Data sharing

Author	DEDIS lab, EPFL
Revision	October 2023 - 1.2.0
Publish date	Friday, October 20, 2023
Due date	**Tuesday, November 14, 2023 @ 23:55**

Table of contents

Introduction **1**
- End result 1
- Objectives 3
- File sharing 3
  - Chunks 3
  - MetaFile and MetaHash 3
- Name and metahash 4
- Catalog 4
- File Download 4
  - Details 5
- Name search “all” 5
- Name search “first” 6
- Search duplicates 6

Your tasks **7**
- Before you begin: New messages introduced 7
  - DataRequestMessage 7
  - DataReplyMessage 7
  - SearchRequestMessage 7
  - SearchReplyMessage 8
- Before you begin: Introduction to the storage interface 8
- Task 0: Think about asynchronous notification 8
- Task 1: Implement the Upload function 9
- Task 2: Implement the catalog-related functions 10
- Task 3: Implement the Download function and process DataReplyMessage 10
- Task 4: Process DataRequestMessage 11
- Task 5: Implement the Tag and Resolve function 11
- Task 6: Implement the SearchAll function and process SearchReplyMessage 11
- Task 7: Process SearchRequestMessage 12
- Task 8: Implement the SearchFirst function 12

Try your program **12**

Appendix A - Tests definitions **13**
- Upload 13
- Catalog 13
- Download 14
- Naming 16
- SearchAll 16
- SearchFirst 18
- Scenario 21
- Integration test 22

Introduction

End result

In this homework you are going to implement a new module for your Peerster related to file sharing[1]. This new module has four main parts: (1) *Data upload*, (2) *Data download*, (3) *Search and index*, and (4) *Naming*. This new module, with functionalities similar to a system like *bittorrent*, is mostly independent from what you implemented in HW1.

The data upload part allows a peer to share a new file on the Peerster system. Once a file is shared (i.e. uploaded) by a peer, other peers can use the download part to get the file. The search and index parts provide the discovery of available files on the Peerster system. This part is needed because peers will have to proactively look for available files. Finally, the naming part is a convenient way to assign a file a custom, easy-to-remember name. This is because, as you will see, files will be stored and referenced by *metahash*, a unique representation of a file based on its content. Figure 1 shows the updated web GUI with the new parts.

A typical workflow will be:

1. Upload a file, which returns a unique identifier M of the file (*metahash*)
2. Download a file providing M (on the same peer where the file was uploaded).
3. Tag an uploaded file with a filename N that maps to M.
4. Execute a search query that will check remote peers about available files. The search result will output available file names.
5. Resolve a file name to the unique file identifier.
6. Download remote files based on the resolved unique identifier.

[1] Note that we sometimes generalize the term “file sharing” to “data sharing”. In fact the implementation is not required to work with files, but uses an abstraction representing data. This is convenient since we can work purely in-memory if needed, which is convenient for testing, or with actual files for a realistic use case.

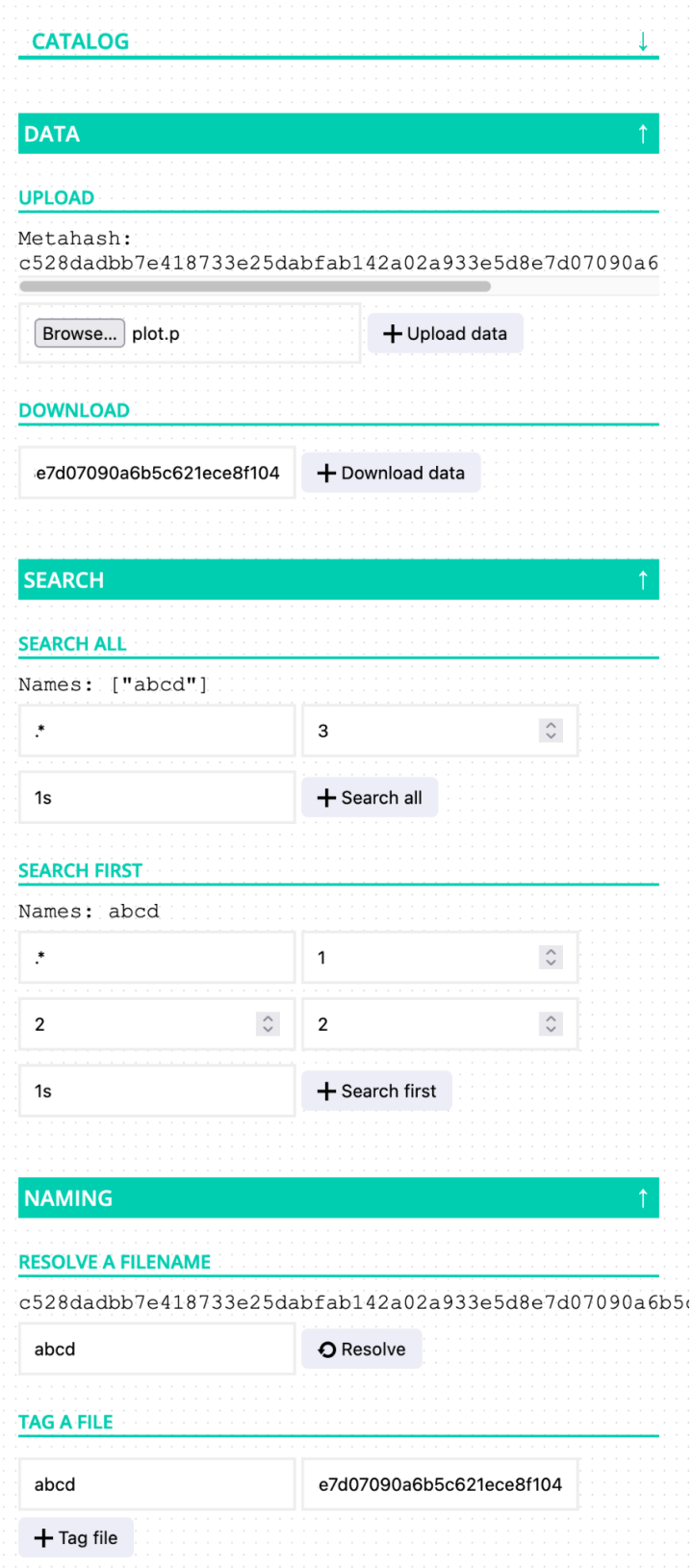

Figure 1: End-result of HW2. The peer is able to upload a file, download a file, perform a search, and tag a file.

Objectives

Implement a new set of functionalities in your peerster related to file/data sharing.

File sharing

Peerster nodes will be able to send and receive potentially large files, not just short text messages. Note that it is already possible (in Homework 1 Peerster) to send large files. Indeed, network layer 2 (Ethernet) would split the packet in datagrams of size MTU (maximum transmission unit), and would reassemble them back at the destination. But, because Peerster uses UDP for inter-node communication, which does not have congestion/flow control, by sending large packets we could flood the network. Moreover, if one part of a large UDP message is lost, the entire message is lost. Thus, peers need to implement their own basic flow control. We'll have peers break up files into chunks for transfer to avoid flooding the network. To keep track of file chunks, peers use Merkle trees (as many P2P systems do), which identify both complete files and parts of files.

The peer indexes each file it shares as follows:

1. It divides the file into chunks and stores each chunk in a Key/Value store using the hex-encoded chunk's hash as the key, and the chunk's content as the value.
2. It creates a MetaFile, which is a file containing the hashes of each chunk and stores it using the hex-encoded MetaFile's hash as the key, and MetaFile content as the value.

Chunks

The chunk size is provided in the configuration. Note that it is very unlikely that the file will be an exact multiple of the chunk size, thus the last chunk may be smaller. This is fine and you must not pad the last file's chunk.

As a simplification, Peerster should only allow sharing files whose MetaFile fits in one chunk. In other words, there's no need to chunk the MetaFile and recursively hash those chunks[2].

[2] We can compute the size of the metafile in bytes as `num_chunks * 64 + Sep_bytes * (num_chunks - 1)`.

MetaFile and MetaHash

To build up the MetaFile corresponding to a shared file, we simply join the 32-byte SHA-256 hash of each chunk with the `MetafileSep` provided.

```
F(x) = Hex_encode(SHA_256(x))
Sep  = MetafileSep
||   = concatenate
c_i  = content of chunk i
N    = total number of chunks
MetafileValue = F(c_0)||Sep||F(c_1)||Sep||...||F(c_N-2)||Sep||F(c_N-1)
MetafileKey = F(MetafileValue)
```

Listing 1: Pseudocode of the Metafile value and key computation. The Metafile value is to be saved using its hex-encoded sha-256 hash content as the key.

As mentioned above, a MetaHash uniquely identifies a file. One consequence is that two files with identical contents but different names have the same MetaHash. Thus, copying a file under a different name, without changing the contents, and sharing it, should not result in creating a different MetaFile. The copy can simply use the same chunks, MetaHash and MetaFile as the original.

For simplicity, Peerster should only support sharing files that are 2 MiB or less.

Name and metahash

When a file is shared, it completely loses its notion of a “filename”. Its unique identifier is its metahash, which is not convenient to remember and use. As such, Peers will maintain a mapping between metahashes and *names*, also sometimes referred to as filename (you can consider both *name* and *filename* as the same). This mapping will be kept in the *naming store*. The naming store is a simple key/value store, where keys are names and values metahashes.

Catalog

Each peer is going to maintain a catalog of metahashes and chunks. A catalog contains, for each known metahash and chunks (referenced by their keys), a bag of peers’ addresses that are known to have them. Note that the catalog itself does not make any difference between a chunk and a metafile. It only has the notion of a “key” which simply says “the value referenced by this key is known to be present on those peers, whether the key represents a chunk or a metahash”. The catalog must only contain information about remote peers, i.e a peer must not have entries about itself.

File Download

To allow a client to download a file, a peer needs to have it locally, or otherwise know a remote peer that has it. Thanks to the catalog this information is at hand. A peer uses a simple one-chunk-at-a-time request / response download protocol:

1. The peer first fetches the metafile from another peer based on its catalog, or locally if it already has it.
2. The peer parses the metafile to extract each hex-encoded chuck hashes, and sequentially retrieve chunks: either locally or from other peers based on the catalog. Recall that a hex-encoded chunk hash is used as the key to reference that chunk value.
3. The peer reconstructs the file with the chunks and returns it.

When the gossiper collects all the chunks for a given file, the gossiper reconstructs it and also takes this opportunity to save the chunks and the metafile locally if some were fetched from a remote peer. Therefore, the files downloaded are then locally available on the peer.

Details

The peer requesting the file first sends a data request to download the MetaFile, if it doesn't have it locally. The node waits for a reply to that request, retransmitting the request periodically if it does not receive a reply. Specifically, the node implements an exponential back-off mechanism with parameters `{I, F, R}`: there are at most `R` retransmissions, each at an interval that starts at the initial timeout `I` that is multiplied by a factor `F` after each iteration.
This means, the node retransmits after `I` seconds from the initial transmission if it does not receive a reply, then again after `I*F` seconds, then `I*F^2`, etc.... Then the node sends a request for each of the file's data chunks in turn (if it does not have the chunk locally), retransmitting the chunk request periodically just like above. For the sake of simplicity, we ask you to download chunks of a file one by one, i.e. when your gossiper sends a request for a chunk, it should wait for the reply before sending the next request. In the real world, several chunks can be downloaded in parallel, however, for the purposes of this homework, the sequential approach must be used.

When a peer receives a request for a MetaFile or a chunk that the peer does not have, the peer responds with an empty value.

Name search "all"

Peers must support regex-based search for available filenames. The search should allow one to discover names on multiple, if not all, peers on the network. The perimeter of the search is limited by a budget, representing the maximum number of peers the search would ask. It works as follow:

1. When a peer searches for a file, it divides the provided budget as evenly as possible among its neighbors and sends a search request to each of them according to the budget.
2. The peer P that receives a search request first processes the request locally, searching among the shared files for any file names matching any of the search keywords, and sending a search reply if any files match.
3. Then, P subtracts 1 from the incoming request's budget. Only if the budget B is still greater than zero, P sends the request up to B of its neighbors, excluding the node that P has just received the request from. If P has more than B neighbors, P forwards the request to randomly chosen B neighbors. If P has fewer than B neighbors, it divides the remaining budget B as evenly as possible (i.e., plus-or-minus 1) among the recipient nodes of this search request.

For example, if an incoming search request has a budget of 3 and the receiving node has 5 neighbors, the node first processes the request locally, subtracts 1 for itself, then forwards the request to 2 other randomly-chosen neighbors, giving each forwarded request a budget of 1. Alternatively, if the incoming request's budget is 10, the node first subtracts 1, then forwards the request to all 5 neighbors, such that 4 forwarded requests have a budget of 2 and the remaining one has a budget of 1.

Name search “first”

Peers need to search for files that can be entirely fetched from a single peer, with no missing chunks. This is what we call a “full match”. In order to not flood the network in search of a “full match”, peers use a simple expanding-ring search[3] scheme with parameters `{B, F, R, T}`. This kind of search is using the same type of message as the previous search, with this time a retry mechanism that increases the budget.

The peer should start with an initial search query budget of `B` and retry a maximum of `R` times. A retry is performed if after the timeout `T` no answer is received with a full match. After each try the budget is multiplied by a factor `F`. The process ends when the peer tried `R` times with no full match response.

Search duplicates

Peers need to detect if they receive a duplicate search request and ignore it. Each request contains a unique identifier that allows peers to identify them.

[3] As you have seen in the lecture “Flooding search and routing”.

Your tasks

In the following tasks you are progressively going to implement the data sharing part, followed by the download part, and finally the searching part.

In summary, here is what needs to be done:

- ☐ 0: Think about asynchronous notification
- ☐ 1: Implement the Upload function
- ☐ 2: Implement the catalog-related functions
- ☐ 3: Implement the Download function and process DataReplyMessage
- ☐ 4: Process DataRequestMessage
- ☐ 5: Implement the Tag and Resolve function
- ☐ 6: Implement the SearchAll function and process SearchReplyMessage
- ☐ 7: Process SearchRequestMessage
- ☐ 8: Implement the SearchFirst function

Before you begin: New messages introduced

Before going into the technical details, here we introduce you to the new types of message Peerster is going to use. Those messages are defined in `types/datasharing_def`.

DataRequestMessage

This message is used by a peer to request a metafile or chunk on a remote peer. It contains a unique `RequestID`, and a `Key`. To get a unique request ID you must use the same function as the one used to generate the PacketID: `xid.New().String()`. The `Key` attribute corresponds either to a metahash or a chunk hash.

DataReplyMessage

This message is sent back by a peer in response to a `DataRequestMessage`. It contains a `RequestID` that corresponds to the `RequestID` from the `DataRequestMessage`, and a Key/Value pair. `Value` can be nil in case the responding peer does not have data associated with the key, otherwise it is set to the value the peer has from the corresponding key. `Key` is the same as the one from the `DataRequestMessage`.

SearchRequestMessage

This message is used by a peer to look for matching filenames on other peers. As for the `DataRequestMessage`, it contains a unique RequestID that must be generated with `xid.New().String()`.

SearchReplyMessage

This message is sent back by a peer in response to a `SearchRequestMessage`. It contains the associated `RequestID` and a list of `FileInfo` that describes a matching filename.

Before you begin: Introduction to the storage interface

You will find a new package `storage` in your skeleton. The storage package contains a `Storage` abstraction. This abstraction is provided to the peer in the configuration and allows you to retrieve two stores: a *DataBlobStore*, to handle the chunks/metafiles, and a *NamingStore*, to handle the mapping between filenames and metahashes. A `Store` is an interface that provides basic data manipulation functionalities expected from a key/value store.

When your peer needs to save a chunk or metafiles, it must use the *DataBlobStore*. For example:

```
blobStore := conf.Storage.GetDataBlobStore()
blobStore.Set(“metahash”, []byte(“chunk”))
```

Your peer can directly use the store for its live operations, without any sort of intermediate structure. Functions provided by the store are thread-safe.

There are two Store implementations: *inMemory* and *file*. As the names suggest, the fist one saves everything in memory while the later one uses files. In the unit tests we use the *inMemory* implementation, and the web GUI will also use by default *inMemory*. You can however use the file storage by providing to the CLI a `--storagefolder` argument. This is a nice way to look at what is being stored, as each key/val pair will be saved as a file in the provided folder. Also, it ensures that a peer doesn’t lose its state between reloads.

Task 0: Think about asynchronous notification

In this homework you will have to deal with asynchronous notification, which resembles the acknowledgement mechanism from HW1. If you couldn’t come up with a satisfying solution, now is a great occasion to think about that, as this homework will require you to implement more of this pattern.

The problem can be stated as follows: Process P1 sends a query and needs to wait (blocking) for a reply to that query. The reply later comes from process P2, which needs to notify P1 of the reply. The query and reply have in common the same unique identifier. In code:

```
func SendQuery() {
  Send query Q
  ■ Wait for reply R_Q
}

func ProcessReply() {
  Receive Rq
  => How to notify ■ ?
}
```

Listing 2: Illustration with pseudo-code of the asynchronous notification problem

Practically, this is what you will need to do for a data request: peer A will send a data request to peer B, and peer A will need to wait for the reply from peer B, which will happen when peer B will process the data reply.

Take some time to think about how to solve this problem. Recall that in your peer implementation you can create as many files and packages as you want.

Task 1: Implement the Upload function

The `Upload` function must create and store in the BlobStore the chunks and metafile. As described in the introduction, split the file into chunks and store each chunk using the hex-encoded chunk's hash as the key. Then save the metafile, which contains all the chunks hashes separated by the `MetafileSep` defined in `peer/datasharing.go`. Use the hex-encoded hash of the metafile as its key and store it too on the BlobStore.

To compute a sha-256 hash use the standard crypto module of go. To hex-encode/decode use the standard encoding/hex module. Listing 1 provides the pseudocode for computing the metafile key and value, and listing 2 provides an example of how to use the Go hash and hex packages.

```
F(x) = Hex_encode(SHA_256(x))
Sep  = MetafileSep
||   = concatenate
c_i  = content of chunk i
N    = total number of chunks
MetafileValue = F(c_0)||Sep||F(c_1)||Sep||...||F(c_N-2)||Sep||F(c_N-1)
MetafileKey = F(MetafileValue)
```

Listing 1 (reminder): Pseudocode of the Metafile value and key computation. The Metafile value is to be saved using its hex-encoded sha-256 hash content as the key.

```
h := crypto.SHA256.New()
h.write(metafile)
metahashSlice := h.Sum(nil)
metahashHex := hex.EncodeToString(metahashSlice)
```

Listing 3: Code example using the crypto and encoding/hex library.

Task 2: Implement the catalog-related functions

Your peer must maintain a catalog that defines where metahashes and chunks can be found in Peerster. The catalog definition and explanations are defined in `peer/datasharing.go`. The catalog is purely local to the peer and doesn't make use of any store. Implement the `UpdateCatalog` and `GetCatalog` functions. Those two simple functions set an entry in the catalog and return the catalog, respectively.

Task 3: Implement the Download function and process DataReplyMessage

The `Download` function must sequentially get the metafile and, based on the metafile content, the chunks. For each of those, the peer must first check if it has the element in its local blob storage. If not, it must select a random peer from the catalog and send it a `DataRequestMessage`, using the routing table to get the next-hop if any.
Note that if the peer doesn't have the element locally and there isn't any entry for the element in its catalog, then the peer must return an error since there is no way to get the element.

Upon sending the `DataRequestMessage` to the random neighbor, the peer must wait for the corresponding `DataReplyMessage` (this is the asynchronous notification problem) and use a backoff strategy in case it doesn't receive the reply back in time. The backoff parameters are given in the configuration. Note that the peer always tries to send the `DataRequestMessage` to the same random neighbor. In case the remote peer responds with an empty value, a tampered chunk, or the backoff timeouts, the function must return an error.

When fetching a chunk, if the remote peer doesn't have the element or sends back the wrong chunk, it means the catalog contains an erroneous element. It should not happen, but if it does for any sort of reason the catalog should be updated to remove that erroneous entry.

You need to implement the processing of `DataReplyMessage` to do the notification needed when fetching an element from the `Download` function. This operation is asynchronous and you can use the `RequestID` to implement some sort of notification. For example, when the `Download` function sends a data request, it could set up a channel identified by `RequestID` that will be filled when a reply is received with the same `RequestID`.

For each element fetched from a remote peer, the peer must store that element on its local blob storage so that it will be available for sharing.

Task 4: Process DataRequestMessage

Upon receiving a `DataRequestMessage`, the peer must respond with a `DataReplyMessage`. This message must have the same `RequestID` and `Key` as the incoming `DataRequestMessage`. The `Value` must come from the peer's blob store. It can be nil if the peer doesn't have the value.

The message must be sent back using the routing table.

Task 5: Implement the Tag and Resolve function

The `Tag` function uses the naming store to create a mapping between a name (sometimes referenced as *filename*) and a metahash. The `Resolve` function resolves a name to a metahash. The naming store must contain names as keys and metahashes as values.

Task 6: Implement the SearchAll function and process SearchReplyMessage

The `SearchAll` function returns a list of all matching filenames from the peer's local naming store and from neighbors. To get filenames from neighbors the function must send a `SearchRequestMessage` to as many neighbors as possible based on the budget. The budget limits the propagation of the `SearchRequestMessage` on the network. It must be divided as evenly as possible among neighbors and randomly allocate budget splitings to its neighbors. For example, if the budget is 10 and the peer has 3 neighbors, then the peer will send a `SearchRequestMessage` with a budget of 4 to a random neighbor, and 3 to the other two. Note that there is no retry mechanism. If the budget is lower than the number of neighbors, then only a random subset of neighbors will receive a `SearchRequestMessage` with a budget of 1.

The function must wait the provided timeout before gathering the filenames from all the `SearchReplyMessage` responses, merged with the peer's local naming store, and returning them (this is again the asynchronous notification problem).

Each `SearchRequestMessage` must be provided with a unique `RequestID`, which will be the same in each responding `SearchReplyMessage`. This is how the link between a reply and a request can be done, as multiple instances of a SearchRequest can occur simultaneously in your peer.

Finally, implement the processing of a `SearchReplyMessage`. There are two elements to implement:

1. Update the naming store and the catalog based on the message content.
2. Notify your peer that a search reply message has been received, based on how you solved the asynchronous notification problem.

Task 7: Process SearchRequestMessage

Upon receiving a `SearchRequestMessage`, a peer must:

1. Forward the search if the budget permits. The forwarded search requests must be sent with a budget evenly distributed among shuffled neighbors. The forwarded request must have all the same attributes of the original request except the budget. The packet's header `Origin` and `RelayedBy` must be set to the peer's socket address.
2. Check its naming store for any matching name and then construct a `types.FilesInfo{}` for each file mapped by a matching name. Include a matching name only if the peer has the corresponding metafile in its blob store.
3. Return a `SearchReplyMessage` containing the same `RequestID` as the request and all the files info. The reply must be directly sent to the packet's source (it can be the peer that originated the search request, or a peer that forwarded a search request) without using the routing table. The Destination field of the packet's header must be set to the `searchMessage.Origin`. If the peer doesn't have any match, it must reply indicating no matches (i.e., an empty slice of responses).

Task 8: Implement the SearchFirst function

When a peer has all chunks of a file, we say that this peer "fully knows" the file. The `SearchFirst` function must return the first matching filename of a fully known file, either locally or by a remote peer. The function must first check if it locally fully knows a filename that matches. For that you will need to resolve the name to a metahash, parse the corresponding metafile, and check if the peer has all chunks. If not, it must use the expanding-ring search to look for a remote peer that fully knows a matching filename. The expanding-ring search's parameters are provided as an input to the function. As for the `SearchAll` function, the peer must send a `SearchRequestMessage`, this time with a budget that increases based on the expanding-ring search's parameter. Remote peers won't make a difference if the search request is for a SearchAll or SearchFirst query. They will simply answer with what they know based on their name store and blob store. It is the responsibility of the initiator to check the search replies and see if one of the replies contains a file that has a total match.

Try your program

As in HW2 and apart from the unit tests and integration tests, you can use the web GUI to try your program. A new set of arguments has been added to the CLI. Use the `-h` flag to see them.

Appendix A - Tests definitions

Upload

Pre-condition	**Action**	**Expectation**
(2-1) P1 with a configuration that has a chunk size of 3. P1 is not started (Start function not called)	Call the Upload function on P1 with a file that is 7 bytes long.	P1 returns the correct metahash and no error. The blob store of P1 must contain the correct metafile and 3 chunks (two chunks of size 3, one chunk of size 1).
(2-2) P1 with a configuration that has a chunk size of 3. P1 is not started (Start function not called)	Call the Upload function on P1 with a file that is 6 bytes long.	P1 returns the correct metahash and no error. The blob store of P1 must contain the correct metafile and 2 chunks.

Catalog

Pre-condition	**Action**	**Expectation**
(2-3) P1, not started	Call GetCatalog on P1 Call UpdateCatalog on P1 with key K and value V Call GetCatalog on P1	Returns an empty catalog Returns a catalog with K and V and length 1

Download

Pre-condition	**Action**	**Expectation**
(2-4) P1, not started	Call Download on P1 with an unexisting metahash for P1	Returns an error
(2-5) P1, not started. The blob storage of P1 has a file with metahash M and chunks CS.	Call Download on P1 with metahash M	Returns a file composed of chunks CS.
(2-6) P1, with no files and no neighbors. P2, with file F associated with the metahash M. P1 has its catalog telling that P2 has F.	Call Download on P1 with metahash M.	Returns an error, because P1 doesn't have P2 in its routing table.
(2-7) P1, with P2 as neighbor. P2, with file F associated with the metahash M. P1 has its catalog telling that P2 has F.	Call Download on P1 with metahash M.	Returns an error, because P2 doesn't have P1 in its routing table.

(2-8) With the following topology: P1 ↔ P2 P1 has its catalog telling that P2 has F. P2 has a file F with associated metahash M and composed of 2 chunks.	Call Download on P1 with metahash M.	Returns F. P1 sequentially sent 3 data requests: one for the metafile, and one for each of the 2 chunks. P1 received 3 data replies corresponding to the data request it sent. P2 received 3 data requests from P1. P2 sent 3 data requests to P1.
(2-9) With the following topology: P1 ↔ P2 File F with metahash M and chunks C1, C2 P1 has the metafile of F and chunks C1 P1's catalog is telling that P2 has chunks C1, C2	Call Download on P1 with metahash M.	Returns F. P1 sent 1 data request for C2. P2 sent 1 data reply with C2.
(2-10) With the following topology: P1 ↔ P2 ↔ P3 File F with metahash M and chunks C1, C2. P1 has the metafile of F and chunks C1 P1's catalog is telling that P3 has chunks C1, C2	Call Download on P1 with metahash M.	Returns F. P1 sent 1 data request for C2. P3 sent 1 reply for C2.

(2-11) With the following topology: P1 ↔ P2, P2 having a DisruptedLayer 100%-duplicator (duplicating all inbound packets) File F with metahash M and chunks C1, C2 P1 has the metafile of F and chunks C1 P1's catalog is telling that P2 has chunks C1, C2	Call Download on P1 with metahash M.	Returns F. P2 received 2 (identical) data requests. P2 sent 1 data reply only.

Naming

Pre-condition	**Action**	**Expectation**
(2-12) P1, not started P1's naming store is empty	Call Tag on P1 with name N and metahash M.	P1's naming store is equal to {N: M}
(2-13) P1, not started P1's naming store is equal to {N: M} (N=name, M=metahash)	Call Resolve on P1 with name N	Return M.
(2-14) P1, not started	Call multiple times Tag on P1 with names Ni and metahashes Mi Call multiple times Resolve on P1 with all the names Ni set previously	Returns the corresponding metahashes Mi corresponding to the names Ni The Catalog is empty

SearchAll

Pre-condition	**Action**	**Expectation**
(2-15) P1, not started	Call SearchAll with pattern `.*` on P1	Returns an empty result
(2-16) P1, not started. P1's name store has a filename N corresponding to metahash M.	Call SearchAll with pattern `.*` on P1	Returns filename N The catalog is empty
(2-17) The following topology: P1 ↔ P2 P2 has an entry in its name store {N:M} but doesn't have in its blob storage an entry for M (i.e. it doesn't possess the file associated to M)	Call SearchAll with pattern `.*` on P1	Returns an empty result, because P2 doesn't have the metafile associated to M.
(2-18) The following topology: P1 ↔ P2 P2 has an multiple entries in its name store: {nameA: mh1, nameB: mh2, nameC: mh3, nameD: mh4, nameE: mh5} P2 has all the metafiles mh{1-5}. Each metafile defines 2 chunks. P2 only has the first chunk.	Call SearchAll with pattern `[name[A-D]]` on P1	P2 sends a reply with 4 responses. Each response says that the file has two chunks but P2 only has the first one. P1's catalog is updated to mention that P2 has mh{1-4} and the chunk. P1's name store is updated with name{A-D} mapping to mh{1-4}

(2-19) The following topology: P1 ↔ P2 ↔ P3 ↔ P4 P1 has {filenameA: mhA} in its name store P2 has {finenameB: mhB} in its name store P2 has {mhB: c1B} in its blob store P3 has {filenameC: mhC} in its name store P3 has {mhC: c1C, c1C: X} in its blob store P3 has {filenameB: mhB} in its name store P3 has {mhB: c1B} in its blob store P4 has {filenameD: mhD} in its name store P4 has {mhD: c1D} in its blob store	Call SearchAll on P1 with a limited budget of 2 and a pattern `.*`	Returns [filenameA, filenameB, filenameC] P4 do not receive any message (not enough budget) P1's catalog is updated as follow: {mhB: {P2:{}, P3:{}}, mhC: {P3:{}}, c1C: {P3:{}}} P1's name store is updated as follow: {mhA: filenameA, mhB: filenameB, mhC: filenameC}

(2-20) The following topology: P1 ↔ P2 ↔ P3 ↔ P4 ↔ P5 P1 has {filenameA: mhA} in its name store P2 has {finenameB: mhB} in its name store P2 has {mhB: c1B} in its blob store P3 has {filenameC: mhC} in its name store P3 has {mhC: c1C, c1C: X} in its blob store P4 has {filenameD: mhD} in its name store P4 has {mhD: c1D} in its blob store P5 has {filenameE: mhE} in its name store P5 has {mhE: c1E} in its blob store	Call SearchAll on P1 with a limited budget of 4 and a pattern `.*`	Returns [filenameA, filenameB, filenameC, filenameD OR filenameE] Since the budget is limited, only one of P4 or P5 should have received a search request. P1's catalog store is updated follow: {mhB: {P2}, mhC: {P3}, mhD: {P4} OR mhE: {P5}} P1's catalog is updated as follow: {filenameA: mhA, filenameB: mhB, filenameC: mhC, filenameD: mhD OR filenameE: mhE}

SearchFirst

Pre-condition	**Action**	**Expectation**
(2-21) P1, not started	Call SearchFirst with pattern `.*` on P1	Returns an empty result
(2-22) The following topology: P1 ↔ P2	Call SearchFirst with pattern `.*` on P1 and an expanding-ring configuration with initial=1, factor=3, retry=2	Returns an empty result P1 sent 2 search requests. The first one with a budget of 1, the second with a budget of 3. P2 sent 2 search replies. P1's catalog is empty P1's name store is empty
(2-23) The following topology: P1 ↔ P2 P1 fully knows a file F with an associated name N	Call SearchFirst on P1 with pattern `.*`	Return N. P1 don't send any search request P1's catalog is empty P1's name store is not updated

(2-24) The following topology: P1 ↔ P2 P1 has {filenameA: mhA} in its name store P2 has {mhA: c1a} in its blob store P2 has {finenameB: mhB} in its name store P2 has {mhB: c1b} in its blob store	Call SearchFirst on P1 with pattern `.*` and a retry=2	Returns an empty result (P1 and P2 do not fully know any file) P1 sent two search requests P1's catalog is updated as follow: {mhB: {P2}} P1's name store is updated as follow: {filenameA: mhA, filenameB: mhB}
(2-25) The following topology: P1 ↔ P2 P1 has {filenameA: mhA} in its name store P2 has {mhA: c1a} in its blob store P2 has {finenameB: mhB} in its name store P2 has {mhB: c1b, c1b: X} in its blob store	Call SearchFirst on P1 with pattern `.* and a retry=2`	Returns filenameB P1 sent 1 search request P1's catalog is updated as follow: {mhB: {P2}, c1b: {P2}} P1's name store is updated as follow: {filenameA: mhA, filenameB: mhB}

(2-26) The following topology: P1 ↔ P2 ↔ P3 P1 has {filenameA: mhA} in its name store P2 has {mhA: c1a} in its blob store P2 has {finenameB: mhB} in its name store P2 has {mhB: c1b<sep>c2b} in its blob store P2 has {c2b: x} in its blob store P2 has {finenameC: mhC} in its name store P2 has {mhC: c1c<sep>c2c} in its blob store P2 has {c2c: x} in its blob store P3 has {filenameC: mhC} in its name store P3 has {mhC: c1c<sep>c2c} in its blob store P3 has {c1c: x, c2c: x} in its blob store P3 has {filenameB: mhB} in its name store P3 has {mhB: c1b<sep>c2b} in its blob store P3 has {c1b: x} in its blob store	Call SearchAll on P1 with pattern `.*`, initial=1, factor=2, retry=2	Returns filenameC P1 sent 2 search request P2 sent 4 packets (2 replies, 1 forward, 1 relay[4]) P3 sent one reply P1's catalog is updated as follow: {mhB: {P2, P3}, c1b: {P3}, c2b: {P2}, mhC: {P2, P3}, c1c: {P2, P3}, c2c: P3} P1's name store is updated as follow: {filenameA: mhA, filenameB: mhB, filenameC: mhC}

[4] The routing table is not used. P3 will directly send back to P2, but the `Header.Destination` will be set to P1. So P2 will relay it to P1.

Scenario

Pre-condition	**Action**	**Expectation**
(2-27) A “complex” topology with 4 peers P{1-4} P1 -> P2 P2 -> P1 P2 -> P3 P3 -> P1 P3 -> P4 P4 -> P2	Call Upload with file Fb on P2. Fb has the associated metahash Mb. Call Download with Mb on P2 Call Tag with name Nb and metahash Mb on P2 Call SearchAll on P1 with a budget of 3 and pattern `.*` Call Download with Mb on P1 Call SearchAll on P3 with a budget of 3 and pattern `Nb` Call Download with Mb on P3 Call Upload with file Fd on P4. Fd hash the metahash Md. Call Tag with name Nd and metahash Md on P4	Returns the metahash Mb Returns the file Fb Returns Nb P1’s name store is updated as follow: {Nb: Mb} Returns the file Fb Returns Mb P3’s name store is updated as follow: {Nb: Mb} Returns the file Fb Returns the metahash Md

	Call SearchFirst on P1 with initial=1, factor=2, retry=5	Returns Nd
	Call Resolve on P1 with Nd	Returns Md
	Call Download on P1 with Md	Returns file Fd
	Add two peers P5 and P6. P6 knows P5 and P5 knows P3. P5 and P6 annonce themselves to the others (heartbeat).	
	Call SearchAll on P6 with pattern `.*` and a budget of 8.	Returns Nb and Nd
	Call SearchFirst on P6 with initial=1, factor=2, retry=4	Returns Nb

Integration test

The integration test is the same as the scenario test, except it uses a reference peer in P2, P3, and P6