Homework 3: File Search and Blockchain Mining

CS438 - Decentralized Systems Engineering 2018

Homework out: Friday, 16.11.2018

Due date: Sunday, 23:59, 09.12.2018

Introduction

In this lab you will enhance Peerster with the following features:

1. A file metadata distribution and keyword search protocol allowing users to search for interesting files on other nodes and download them.
2. A blockchain to provide a common mapping name-to-file to allow users to uniquely name a file and ease the file download process.

Hand-in Procedure and Grading

For each homework, you will receive a grade out of 6, as according to the EPFL grading system. To be considered for receiving full points (max. 6), you must upload and submit your fully-working code on Moodle (simply as a collection of source files) before the due date. **You can always update your submission on moodle, so please start submitting early. You wouldn't want a few minutes delay to cause you to miss the deadline.**

We will grade your solutions via a combination of testing and code inspection (and code plagiarism detection). We will run automated tests on your application and make sure it works as required, both when communicating with other instances of itself and when communicating with our own instances. Manual code inspection will mainly come into play when evaluating that you have implemented GUI and its communication with the backend correctly and also to verify that you have implemented all the required techniques.

Our very first test is that your code compiles when `go build` is executed on your files!
No points will be given if your code does not compile.

We will not dock points merely for stylistic deficiencies or ugly hacks—although we strongly encourage you to try to keep your code clean and maintainable, because you will have to keep building on it throughout the semester, and design flaws that you manage to work around in one lab may well come back and bite you in the next.

You're responsible for writing your own test script, you could take inspiration from the first assignment ;)

Part 1: File Search

Now that we can share and download files, we need a more convenient way to search for interesting files to download. For this purpose we will implement a simple expanding-ring flooding scheme.

Exercise 1

Add a GUI and CLI option to search for a file by keyword, by allowing the user to enter one or more keywords to search for files at nearby nodes whose filenames contain any of those keywords, **using as regular expression `*keyword*`, where `*` represents 0 or more characters**. Your GUI and CLI code should simply accept a line of text, and treat comma characters as keyword separators: `keyword1,keyword2,etc.` You will then need to add two new message types to the protocol to handle searches:

```
type SearchRequest struct {
	Origin   string
	Budget   uint64
	Keywords []string
}

type SearchReply struct {
	Origin      string
	Destination string
	HopLimit    uint32
	Results     []*SearchResult
}

type SearchResult struct {
	FileName     string
	MetafileHash []byte
	ChunkMap     []uint64
	ChunkCount   uint64
}

type GossipPacket struct {
	Simple        *SimpleMessage
	Rumor         *RumorMessage
	Status        *StatusPacket
	Private       *PrivateMessage
	DataRequest   *DataRequest
	DataReply     *DataReply
	SearchRequest *SearchRequest
	SearchReply   *SearchReply
}
```

When a Peerster node receives a search request, it first processes the request locally, searching the names of the files stored locally for any files matching any of the search keywords, and sending a search reply as described below if any files match. Then, the node subtracts 1 from the incoming request's budget, then only if the budget B is still greater than zero, redistributes the request to up to B of the node's neighbors, subdividing the remaining budget B as evenly as possible (i.e., plus-or-minus 1) among the recipients of these redistributed search requests.

For example, if an incoming search request has a budget of 3 and the receiving node has 5 neighbors, the node first subtracts 1 for itself, processes the request locally, 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.

Important! Peers need to detect if they receive a duplicate `SearchRequest` and ignore it. Duplicate means the `Origin` and `Keywords` are the same as in another `SearchRequest` **received in the last 0.5 seconds**.

You can assume all SearchReply messages correspond to a previously-issued search request, which means there are no unsolicited or malicious SearchReply messages.

A search reply is a point-to-point message containing the usual `Destination`, `Origin`, and `HopLimit` fields, as well as an array of `SearchResult`. A SearchResult contains the name of the file that matched the keyword search, the hash of its metafile, and `ChunkMap` as the indexes of the chunks that the replying peer contains locally, e.g., [1,4,7], **and finally the number of chunks of the file (**`ChunkCount)`.

When the user performs a search, if the budget is not specified, your node should start with a search query budget of 2, then periodically (once per second) repeat the query, doubling the budget each time, until you reach some maximum budget (32) or obtain some threshold number of total matches (**for our tests, please use a match threshold of 2**). **A match means all the chunks of the matched file are present at at least one peer.** Recall the number of chunks of file carlton.txt, for example, is available in its metafile (**and in the SearchResult**), and any peer that has a chunk of file carlton.txt also has the metafile of carlton.txt. You can get the metafile of a file at random from any peer that has a chunk of that file. In the GUI, you should display the matches obtained in a list as you receive them, in the order the matches are received, so that matches received earlier show up at the top of the list. The user should then be able to double-click on a file to download it, by initiating the download protocol above to the appropriate node (identified by the Origin in the search reply for the hit). If chunks of a certain file are present at multiple peers, then you can download them from random peers.

As a command-line client, you should perform file search, as well as file download. The command for file search is as follows:

```
./client -UIPort=8080 -keywords=share,ex -budget=2
```

`budget` is an optional parameter. If it is missing, then the gossiper starts with a budget of 2 and increases it as described above.

Downloading the metafile for each search result is not mandatory if you choose to use the `ChunkCount` information inside the SearchResult. In this case, please disregard the "DOWNLOADING metafile" lines in the example below.

Output example:

An output for `./client -UIPort=8080 -keywords=share,ex` could be (assume sharedFile.txt has 4 chunks, 1ex23.txt has 2 chunks and ex.txt has 7 chunks; colors are simply for helping to follow the output)

```
FOUND match sharedFile.txt at node12
metafile=ccd6ce0ac6a319cb4b9f72a6e1958136333a657c1f5b6f9756cefe6992be
5318 chunks=2
FOUND match ex.txt at node3
metafile=abc6ce0ac6a319cb4b9f72a6e1958136333a657c1f5b6f9756cefe6992be
5123 chunks=7
DOWNLOADING metafile of sharedFile.txt from node12
FOUND match ex.txt at node4
metafile=abc6ce0ac6a319cb4b9f72a6e1958136333a657c1f5b6f9756cefe6992be
5123 chunks=2,3
FOUND match 1ex23.txt at node5
metafile=6e1950ac6a319cb4b9f72a6e1958136333a657c1f5b6f9756cefe6992be5
1234 chunks=1,2
DOWNLOADING metafile of 1ex23.txt from node5
DOWNLOADING metafile of ex.txt from node4
FOUND match sharedFile.txt at node77
metafile=ccd6ce0ac6a319cb4b9f72a6e1958136333a657c1f5b6f9756cefe6992be
5318 chunks=1,2,3
FOUND match sharedFile.txt at node33
metafile=ccd6ce0ac6a319cb4b9f72a6e1958136333a657c1f5b6f9756cefe6992be
5318 chunks=2,4
FOUND match ex.txt at node43
metafile=abc6ce0ac6a319cb4b9f72a6e1958136333a657c1f5b6f9756cefe6992be
5123 chunks=1,2
FOUND match 1ex23.txt at node77
metafile=6e1950ac6a319cb4b9f72a6e1958136333a657c1f5b6f9756cefe6992be5
1234 chunks=2
SEARCH FINISHED
```

Explanation: the gossiper increased the budget up to 4, and with a budget of 4 found all chunks for 2 matched files (recall we use a threshold of 2): 1ex23.txt and sharedFile.txt. Because it already found all chunks for 2 files, it didn't increase the budget further, to find all chunks of ex.txt. Files 1ex23.txt and sharedFile.txt are thus available for download.
Note there is an end-of-line character (`'\n'`) only between the lines starting with 'FOUND', 'DOWNLOADING' and 'SEARCH'.

Output format

- When receiving a search reply from peer named <peer> print at stdout for each matching search result:

```
FOUND match <filename> at <peer> metafile=<metahash> chunks=<chunk_list>
```

Where <filename> is the FileName of the SearchResult, <metafile> is the MetafileHash of the SearchResult, and <chunk_list> is the comma-separated list of chunks "indexes" in the SearchResult's ChunkMap field, sorted in ascending order. The first chunk is the chunk with "index" 1.

- When two full match are detected for a search, print:

```
SEARCH FINISHED
```

Then download a matched file as in homework 2, but this time do not specify -dest. The gossiper needs to keep state after the search command to remember which peers have which chunks. Recall that reconstructed files should be stored in `hw3/_Downloads/`.

```
./client -UIPort=8080 -file=sharedFile.txt
-request=ccd6ce0ac6a319cb4b9f72a6e1958136333a657c1f5b6f9756cefe699
2be5
```

```
DOWNLOADING sharedFile.txt chunk 1 from node77
DOWNLOADING sharedFile.txt chunk 2 from node12
DOWNLOADING sharedFile.txt chunk 3 from node77
DOWNLOADING sharedFile.txt chunk 4 from node33
RECONSTRUCTED file sharedFile.txt
```

General remarks

To test your search and downloading code effectively, set up a test topology containing several nodes connected only indirectly, and make sure you can search and download across indirectly connected nodes. Put several files on each node, and make sure the filenames have both "common" and "rare" keywords as substrings. When you search for a keyword matching several files, make sure that matches at "nearby" nodes tend to appear first (as they should due to the expanding ring search), but that you eventually see all the matches (e.g., after several seconds as the search budget increases to encompass the entire test network).

This completes the first part.

Part 2: Naming with a Blockchain

Now that we can search for files, we need a human-readable identifier for each file instead of using the metafile hash. We also need this name-to-metahash mapping to be global and agreed on so that we can protect from adversarial peers who serve viruses instead of the correct file. As usual, blockchain is the solution.

Exercise 1

In this exercise, you will implement (1) claiming the mapping of file names to their metafile hashes, (2) processing the claims of other peers, and (3) publishing the accepted mappings to a blockchain.

You should start by creating name binding `TxPublish` to the gossiping mechanism which would represent a transaction to be published (see the struct specification below). So now, whenever you add a new file to your Peerster to be indexed and potentially shared, you need to notify other peers in the network about this exciting event by sending out a corresponding transaction. For simplicity, we ask you to broadcast transactions to all the connected peers but to also set `HopLimit` to your transactions (use 10). Transactions are broadcast using regular `GossipPacket`. Upon receiving a `TxPublish`, a peer checks if it has already seen this transaction (since the last mined block) and that the transaction is valid (i.e., nobody has already claimed the name). If it is a valid transaction and has not yet been seen, it is stored by the peer in order to be included in the next block to be mined, and it is broadcast to other connected peers after `HopLimit` decrementing. The transaction is discarded if either condition is not met. `HopLimit` does not to be as large as the width of the network because peers do not need to receive all the transaction candidates. They will learn the published transactions via blockchain instead!

In a decentralized system, it is not easy for participants to agree on some common state, in our case on the history of claimed name-to-metahash mappings. Peers could split history into chunks, or *blocks*, and agree on them one by one. But if peers see transactions at different time due to the propagation delay and are likely to have a slightly different set of transaction candidates at hand at a given moment, how do they agree on what should be published? One way to solve this would be to elect a one-time leader who would propose the transactions to be included in the next block, and the other peers would verify the validity of this proposal. But how can peers elect such a leader repeatedly?

Proof-of-Work (PoW) is a probabilistic protocol for leader election which uses randomness for decision making. To participate in it, the peers become miners who try to publish blocks of transactions. A block is identified by its hash and it contains(1) the hash of the previous block (parental block), (2) a nonce value, and (3) the transactions to be validated. The idea is pretty simple: the miner chooses transactions to include in the new block, identifies the hash of previous block and generate a random nonce. Then, the miner calculates the hash of the block and checks whether this hash starts with a certain number of zero bits (16 zero bits in our case). If this is true, we say that a block is mined, or alternatively, the miner becomes a leader who can propose the block to other peers. If the block's hash does not satisfy the requirement, the miner can choose a new nonce and try again. When a miner finds a valid block, it broadcasts the block to the network in the same manner as TxPublish are flooded (drop if your peer has already seen it), but using a HopLimit of 20.

Nodes that receive a new block check: (1) if they have already seen the parent block, (2) if the PoW is valid. If these two conditions hold, then they accept this block, append it to their blockchain, and integrate the transactions in the block into their name-to-metafile mapping.

In the real world, the start of a blockchain is indicated by a genesis block that every node needs to obtain, and either mine directly on top of it or verify the connection to the blocks and then mine on top of them. But for the sake of this exercise, you should 0 as `PrevHash` in mining a new block if your peer just joins the network, and you should accept `PrevHash` of another peer's block as valid if it is the first broadcast block that you receive after joining.

For this exercise, you will use the following new message types:

```
type TxPublish struct {
      File     File
      HopLimit uint32
}

type BlockPublish struct {
      Block    Block
      HopLimit uint32
}

type File struct {
      Name          string
      Size          int64
      MetafileHash  []byte
}

type Block struct {
      PrevHash     [32]byte
```

```
        Nonce        [32]byte
        Transactions []TxPublish
}

type GossipPacket struct {
        Simple        *SimpleMessage
        Rumor         *RumorMessage
        Status        *StatusPacket
        Private       *PrivateMessage
        DataRequest   *DataRequest
        DataReply     *DataReply
        SearchRequest *SearchRequest
        SearchReply   *SearchReply
        TxPublish     *TxPublish
        BlockPublish  *BlockPublish
}
```

You will also need to use these (exact) implementations of hashing in order to be compatible with other implementations of the blockchain:

```
func (b *Block) Hash() (out [32]byte) {
        h := sha256.New()
        h.Write(b.PrevHash[:])
        h.Write(b.Nonce[:])
        binary.Write(h,binary.LittleEndian,
uint32(len(b.Transactions)))
        for _, t := range b.Transactions {
                th := t.Hash()
                h.Write(th[:])
        }
        copy(out[:], h.Sum(nil))
        return
}

func (t *TxPublish) Hash() (out [32]byte) {
        h := sha256.New()
        binary.Write(h,binary.LittleEndian,
uint32(len(t.File.Name)))
        h.Write([]byte(t.File.Name))
        h.Write(t.File.MetafileHash)
        copy(out[:], h.Sum(nil))
        return
}
```

The testing system will be looking for the following lines of output from your Peerster:

- Each time that a new block is integrated onto the head of your chain, you should emit one line of the form
  `CHAIN [block-latest] [block-prev] … [block-earliest]`
  Blocks are colon-separated sets of the block hash, the previous block hash, and the transactions. The transactions are comma-separated lists of filenames. While you are working on exercise 1, CHAIN lines from different servers will diverge, and for any one server, the CHAIN line will monotonically increase in size.
- Each time that your miner finds a block, it should emit one line of the form
  `FOUND-BLOCK [hash]`
  where hash is the hex-encoded form of the found block hash.

Exercise 2

Creating and gossiping blocks does not necessarily mean that the peers are going to reach consensus on the history of the blockchain. As we discussed in the class, in Bitcoin, there are times when several peers become one-time leaders by mining valid blocks concurrently, which creates multiple versions of the blockchain (i.e., forks). In the second exercise, you will implement Bitcoin’s consensus algorithm, called Nakamoto consensus.

Until now, we assumed that when a valid block arrives, we persist it by directly updating the state with the transactions in the block. In this exercise, you will extend your implementation to maintain concurrent chains. When a block arrives with a valid PoW, the peer looks into its current blockchain to check whether it is extending the longest chain or not. If it is extending the longest chain, then the protocol in Exercise 1 suffices. If it is not, then it needs to store the block locally (without applying the state updates), and to check whether appending the new block converts a previously deprecated fork to the longest chain. If this happens, then the peers first roll-back their current longest chain (by deleting the transactions inside the block) until the first block that is a common ancestor of both chains. Afterwards they start following the new longest chain as described in Exercise 1. Once they get up-to-date, they can restart the mining process. Your blockchain state might happen to look as in Figure 1 indicating the longest chain (blue) and the two shorter chains (green, orange). Note that the length of the orange chain is only one block less than of the blue, as they share three common blocks.

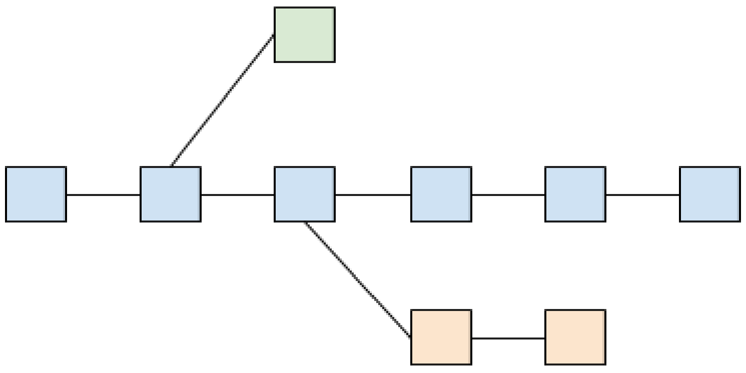

Figure 1

You will find it much easier to create (and resolve) forks if your miners record how long it took them to find a block and then sleep twice the mining time before publishing the block, in order to give time for other miners to also find a valid block. In Bitcoin, implementing such behavior would be a terrible idea, because your miner would disadvantage itself with respect to all the other miners in the system. But in this homework, it causes many more forks, which is much more… exciting!

The testing system will be looking for the following lines:

- FOUND-BLOCK, as before.
- CHAIN will be emitted each time that Peerster updates its idea of what the current longest chain is. If consensus is working correctly, the CHAIN lines from different servers will converge, but their first few blocks could remain out of sync.
- Each time your Peerster is required to undo some transactions in order to follow the new longest chain you should emit

  FORK-LONGER rewind %d blocks
- Each time your Peerster discovers a fork which creates a shorter chain, you should emit

  FORK-SHORTER hash

  where hash is the block hash of the block that links to the front of a shorter chain.