CPSC 426 Lab 3: Decentralized Search and File Sharing

Handed out Monday Sep 30, 2013
Bakeoff version due Friday October 4, 2013 11:59PM
Final version due Friday October 11, 2013 11:59PM

Introduction

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

1. A simple file-sharing protocol enabling nodes to copy files between each other.
2. A file metadata distribution and keyword search protocol allowing users to search for interesting files on other nodes and download them.

Development and Hand-in

For this and subsequent labs you should keep working in, and committing changes to, the same Git repository you checked out for lab 1. Since Git records all history in an "append-only " log, you will always be able to roll back to previous versions for debugging or other reasons any time you need to.

As before, hand in your completed bakeoff version and final version, by the respective deadlines above, by using `git commit` and `git push` to push your changes to your master repository in the SUBMIT directory.

Don't forget to `git pull` your handed-in version to your `peerster-test` repository or another clone of the repository to make sure the submitted version is complete and actually works! Since shopping period is over, points will be deducted for late or incomplete assignments as explained in the course overview

Part 1: File Sharing

We will now give Peerster nodes the ability to send and receive potentially large files, not just short text messages. Since Peerster uses UDP for everything and UDP works reliably only with short (e.g., up to 8KB) datagrams, we will have to break files up into blocks for transfer. Nodes will also need to be able to identify files to each other to announce them and ask to download them. As in many P2P systems, we will use hash trees to identify both complete files and parts of files (Wikipedia page). Since in this lab we will start using cryptographic algorithms as building blocks, we will need a suitable crypto library. Although there are many crypto libraries available that offer what we need, we recommend that you use QCA, the Qt Cryptographic Architecture, since it is designed to be well-integrated with Qt:

- Home page
- API documentation

To share files, the first thing you will need to do is allow the user to specify one or more files to be shared, and to index and divide those files into blocks.

Exercise 1. Add a "Share File..." option (e.g., button) to the UI, which opens a file selection dialog box (`QFileDialog`) that allows the user to select one or more files to share. The user should be able to shift-click to select multiple files without having to open the dialog box multiple times.

Challenge: Optionally, you might want to add a feature such that if the user picks a directory to share, your Peerster recursively shares all files in that directory and any descendant subdirectories.

Now we need to do something with the files the user chooses to share: initially, just scan and catalog them internally.

Exercise 2. Now, for each file the user chooses to share via the GUI dialog above, you will need to scan the file, divide it into blocks, and compute a SHA-256 hash on the contents of each block. For simplicity we will just use a fixed 8KB block size.

As a result of scanning each file you'll need to build up a *blocklist* metafile, which is simply a file containing the SHA-256 hashes of each block. To form the blocklist file, simply concatenate the 32-byte SHA-256 hashes of each file block into one large `QByteArray`, so that the blocklist for an *N*-block file results in a *32×N*-byte blocklist metafile. Once you have this blocklist metafile, compute its SHA-256 hash and store that as well as the file's *blocklist hash*.

After scanning all the files to be shared, your Peerster should have the following metadata in its internal data structures, for each file:

- File name on the local machine.
- File size in bytes.
- The blocklist metafile computed as described above.
- The SHA-256 hash of the blocklist metafile.

Challenge: With the above format, any file above 2MB in size will yield a blocklist metafile larger than 8KB, violating our block size. Optionally, you may wish to add support for files larger than 2MB by recursively applying this procedure: e.g., if you have a first-level blocklist metafile that's larger than 8KB, divide this metafile into 8KB blocks and produce a second-level blocklist metafile containing the hashes of the blocks of the first-level metafile, and so on for as many levels as needed to handle arbitrary-size files.

Challenge: You might wish to add support for dividing files into blocks using an rsync-style content-aware blocking algorithm, using a rolling checksum scheme such as Adler-32, which can make block caching and deduplication more effective. If you

do this, please enable variable-length block behavior *only* if a suitable command-line option is given (e.g., '`-varblock`'), since we don't expect everyone's Peerster implementations in this course to be able to deal with variable block sizes. If you do this, please state in your implementation notes (e.g., README file) what algorithm you used and how to enable variable-length blocks.

Finally, we need a protocol for one node to retrieve a file from another node. For now we will assume the retrieving node already has the hash of the desired file's blocklist metafile (which we'll call the *file ID*); the search scheme in the next part will provide a way for nodes to learn this file ID hash. We will use a simple one-block-at-a-time request/response download protocol.

Exercise 3. Add support for two new messages to your protocol, for block requests and block replies:

- **Block request:** This is a message whose `QVariant` contains a "`Dest`" key indicating the node identifier two whom the block request is targeted; a "`Origin`" key indicating the identifier of the node that sent the request; a "`HopLimit`" key whose `quint32` value limits the number of hops over which the request may be routed before being dropped; and a "`BlockRequest`" key whose value is a `QByteArray` containing a SHA-256 hash of the block requested. This hash can be either the hash of a "regular" file data block, or the hash of a blocklist metafile.

- **Block reply:** This message has the same `Dest`, `Origin`, and `HopLimit` keys as above, plus a key "`BlockReply`" whose value is the SHA-256 hash of the block this reply contains, and fianally, a key "`Data`" containing the actual content of that block as a `QByteArray` value. For the block reply to be valid, the SHA-256 hash of the `Data` value should always match the explicit hash carried in the message's `BlockReply` field. Your code should check this invariant when it receives a block reply and drop the message if the message contains incorrect data.

Also add a simple GUI button to download a file from another node. To start with, you can simply have the user enter a target node ID and a hexadecimal blocklist metafile hash to serve as the file ID. The node requesting the file will first need to send a block request to download the blocklist metafile, wait for a valid reply to that request (with a hash matching the requested hash and matching content), retransmitting the request periodically as necessary, and then download each of the file's data blocks in turn. You can keep the protocol simple by requesting only one block at a time during a given file download.

Peerster nodes must route these messages "point-to-point" exactly as described in Part 1 of Lab 2, so that a node can download a file from another node that is connected only indirectly via intermediate hops. Note that these new messages both have `Dest` and `HopLimit` fields, just like point-to-point text messages, which are the only fields the routing logic needs to care about. Thus, your code can decide whether to route a message on to another node or to process it at the local node simply by looking for

(and at the contents of) these fields before doing any local processing. You need to distinguish between text messages and block requests/replies only after the message has reached its destination (i.e., the message's `Dest` field refers to "me").

Challenge: Requesting only one block at a time as suggested above can be slow, because we pay a full network round-trip delay for each block, and are unlikely to use more than a tiny fraction of the available network bandwidth this way on most real networks. But we don't just want to request "all" a file's blocks at once either, as a barrage of requests such as this would likely overload the network and cause excessive packet loss. If you want to make your downloads go fast in a "responsible" way, implement a simple TCP-like congestion control scheme, for example along the lines of Van Jacobson's scheme as described in Van Jacobson's seminal paper, and more recently in RFC 5681. That is, increase the number of concurrent block requests you allow until you start noticing losses (of either requests or replies), and use the arrival of replies to "clock" the sending of new requests just like in TCP.

Please note that there are three minor clarifications about our Lab3:

- QtCrypto, which you will need, is already part of the Qt distribution installed on our Zoo; however, in order to enable it, you need to add `CONFIG += crypto` to your 'peerster.pro' file.
- Once you start trying to use `QtCrypto`, you will get a mysterious assertion failure unless you first initialize it by adding a line `QCA::Initializer qcainit;` to your `main()`
- When you are breaking files into 8KB blocks, in most cases the file will of course not be a natural multiple of 8KB in size; in this case your last block will be less than 8KB in size. Do NOT pad the last block to 8KB, since if you did so you would "lose" the file's correct original size.

Part 2: 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 4. Add a GUI option to search for a file by keyword, by allowing the user to one or more keywords and then searching for files at nearby nodes whose metadata (i.e., filenames) contains any of those keywords. Your GUI code should simply accept a line of text, and treat space characters as keyword separators. You will then need to add two new message types to the protocol to handle searches:

- **Search request:** A search request has a "`Origin`" key whose `QString` value indicates the node that sent the search message; a "`Search`" key whose `QString` value is the search string the user entered (i.e., a list of keywords separated by spaces), and a "`Budget`" key whose `quint32` value indicates the search request's "rebroadcast budget".

  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 search 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, f 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.

- **Search reply:** A search reply is a point-to-point message containing the usual `Dest`, `Origin`, and `HopLimit` fields, plus a "`SearchReply`" key whose `QString` value is the exact search string of the search request this reply is associated with; a key "`MatchNames`" key whose value is a list (`QVariantList`) of filenames (`QString`) matching the search query; and a "`MatchIDs`" key listing the corresponding file IDs (blocklist metafile hashes encoded as `QByteArray`) of the matching files, in the same order.

When the user performs a search, your node should start with a search query budget of 2, then periodically (e.g., once per second) repeat the query, doubling the budget each time, until you reach some maximum budget (e.g., 100) or obtain some threshold number of total matches (say, 10). 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).

Challenge: Add support for more types of file metadata in searches: e.g., file size, modification date, maybe a user-entered description. If you're adventurous, use one of the various libraries available online to look for and parse ID3 tags in MP3 files the user is sharing.

Challenge: Add support for more complex types of searches: e.g., recognize the special words "and" and "or", to allow users to specify searches like "`foo` ***`and`*** `bar`", or "`foo` ***`or`*** `bar`", and treat them appropriately.

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 lab.