CSC 2209: CLOUD STORAGE FINAL PROJECT

DAVID SOLYMOSI AND JIMMY ZHU

1. High Level Overview

We implemented a backup and sync service with a focus on minimizing network traffic at the cost of local storage and computation. The files for the cloud client and server applications can be found at:

`http://www.cs.toronto.edu/~solymosi/cloud/`.

The client application was responsible for keeping files on the local machine up-to-date with the server. This was accomplished by monitoring the files for local changes and conveying these changes to the server, as well as querying the server for changes and updating the local files as necessary. To minimize network traffic, instead of sending the entire updated file, the client sent the changes via a diff file whenever possible.

The server application was responsible for storing the most up-to-date versions of the files being monitored. This was accomplished by listening to the clients for any changes and implementing the changes received to the files stored on the server. The server also responded to clients' requests for newer versions of the files. As before, to minimize network traffic, the server sent diff files whenever possible.

The performance of our implementation was significantly better for large text files when there were only small changes as opposed to transferring the entire file. Due to space constraints, our experiments cannot be included in this report.

2. Implementation Details

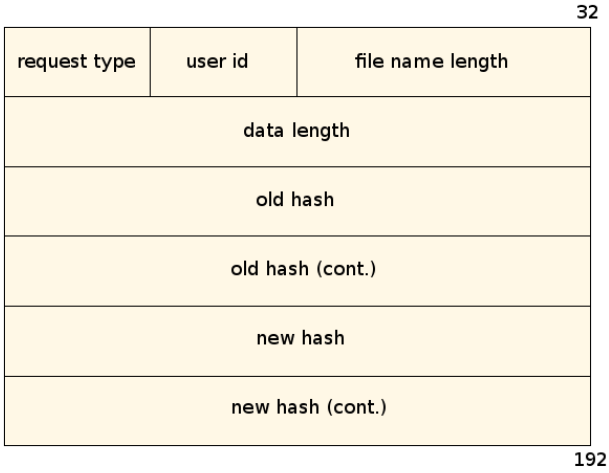
Our implementation of the cloud storage system was for Linux only, and relied heavily on the workings of `patch`, `diff`, and `combinediff`. TCP was used, and some details such as port number were hardcoded for this demonstrative version.

2.1. **Protocol.** TCP messages with a wide range of sizes need to be sent for cloud storage to function properly, so we precede every message by a header which is defined the same way for the client and the server, within the `cloud_protocol.h` file. This header tells both the client and server the purpose of the message, the size of the data fields accompanyimg the message (if any), and other useful pieces of information about about the contents. See Figure 1 for a visual representation of the header.

The cloud header starts off by specifying an 8 bit request type, that is, what is the purpose of this message. The next field is an 8 bit user identification field which can be used to distinguish users. This is followed by a 16 bit unsigned integer corresponding to the file name length which is included with the message. Following this is a 32 bit unsigned integer corresponding to a data length which is

Date: June 13, 2016.

FIGURE 1. Layout of the cloud header.



also included with the message. Next are two 64 bit hash values, which can be used as identifiers of the file the message is about. Their use will be explained below.

The header is followed by the specified number of bytes of the file name, which is followed by the specified number of bytes of the data. A typical message will be $24 + n + m$ bytes long, where n is the length of the file name in question, and m is the number of bytes that is needed to be sent.

Each message is one of the following types:

- `req_update`, which is used to send patches when local files are updated. It contains the filename and the patch data, and the two hash fields are used to specify what old version of the file is being patched to what new version of the file.
- `req_updatefull`, which is used to send full local files. The filename is sent, as well as the full contents of the file as data. The old hash field is unused, but the new hash is the hash of the file sent.
- `req_reqpatch`, which requests a patch for a given filename. The data length should be 0, and no data should be included after the file name. The patch expected in return is the one that would get the file from the included old hash to the included new hash.
- `req_reqfull`, which requests a full file for a given filename. The data length should be 0, and no data should be included after the file name. The old hash and new hash fields are unused, and the newest version of the file is expected in return.
- `req_delete`, which indicated that a file has been deleted locally, and should be deleted at the destination. The data length should be 0, and no data should be included after the file name. The old hash and new hash fields are unused.
- `req_uptodate`, which is a confirmation message stating that a file with the given filename is the most recent one. The hash of this newest file is included in the new hash field, but the old hash field is unused. The data length should be 0, and no data should be included after the file name.

- `req_checkall`, which is a special type of message. File name length is unused, and no file name is specified. The old and new hash fields are unused. The data length is provided, and the data has a very specific structure. It consists of a series of concatenated messages, which contain a 16 bit unsigned integer corresponding to file name length, a 64 bit hash field, and a file name over the number of bytes which was specified.

  The purpose of this message is to let the other party know what files you have locally stored, along with their hash values (versions), so that some synchronization can follow.
- `req_notify`, which is used to notify the other party that the synchronization following a `req_checkall` type message is complete. Aside from the request type and user identification, all other fields are unused, but the file name length and data length should be zero. No data should follow the header.

2.2. **Server.** The role of the server is to store the newest versions of files, as well as to be able to respond to requests from any client. In particular, it must respond correctly to the various requests and sent files and patches, as well as other request types.

The server listens for incoming connections and messages on a single port. When a connection is established by a client, the server waits for and parses messages, each of which is expected to start with the cloud header. How the server deals with each message will be detailed below.

If the client closes the connection, or the client does not send new messages for a long time, the server goes back to waiting for connections, ready to accept one from the same or any other client.

The structure of the server is straightforward. The working directory for the server will be `.cloudserver`, which will contain a stored list of hash values in a file `.crcfile`, as well as a directory for each active file that has been synced to the server by a client previously.

Each directory of a file is guaranteed to include the file `curr`, which is the latest version of the file. The hash value of this file is the one that is stored in `.crcfile` one directory above. The folder might also contain other files; each other file will have a name that corresponds to a hash value. The contents of each of these files will be a patch that updates a previous version of the file (the version whose hash was the filename) to the latest version (which is `curr`).

The server deals with each type of request differently:

- When a `req_update` is received, we are being sent a patch. We check to see that this patch can be applied to the latest file with the given filename, and if it is, we apply it to `curr`, combine all the existing patches in the folder with this patch, and store it as a patch to get from the version we had just before getting the message to this new one. We update our `.crcfile` to hold the hash of this newest version.

  If the patch we receive cannot be applied, we reply with a `req_reqfull` for the file specified.
- When a `req_updatefull` is received, we are being sent a file. If we already have a saved version of this file, we create a `diff` file from that one to this received version, and proceed as in the previous case.

If we do not have a local version of this file, we save it as the `curr` file, and update our `.crcfile` to include the hash of this file.

- When we receive a `req_reqpatch`, we will reply with a `req_update` containing the patch at `$filename/$oldhash`.

  If that file doesn't exist, we send a `req_updatefull` containing the file `$filename/curr`.
- When we receive a `req_reqfull`, we will reply with a `req_updatefull` containing the file `$filename/curr`.
- When we receive a `req_delete`, we will delete the entire folder of the file specified, and we remove the entry for the file in `.crcfile`.
- When we receive a `req_uptodate`, we must have done something right! So we move on happily.
- When we receive a `req_checkall`, we will loop though the file names and hashes given in the data part of the message.

  For each message, we check if we have the file with the given filename. If we don't, we send a `req_reqfull` for the file. If we do, we check if the hash matches with ours. If it does, we send a `req_uptodate`. If it doesn't, but we have a patch to get from that version of the file to our newest version of the file (i.e. `$filename/$hash` exists), we send a `req_update` with that patch. If it doesn't but we don't have a patch, then we send a `req_reqpatch` to get from our newest version of the file to their version.

  Finally, for each file that we have but was not included in this list, we send a `req_updatefull` message containing the file.
- We should not receive any `req_notify` messages, so we ignore any incoming messages of this type.

2.3. **Client.** The role of the client is to keep the local files up-to-date with those stored on the server. In particular, it must be able to detect changes within local files, communicate these changes to the server, as well as request more up-to-date files from the server. On the surface, these seem like easy tasks but, as we will see, many subtle problems may arise which required us to make assumptions on what the client may or may not do. We will discuss how to relax and avoid these assumptions in a later section.

To detect changes, we keep a separate directory from the main sync directory, called the replicate directory. In the replicate directory, we store the last synced version of all the files with the server. The main idea is to guarantee that the files in the replicate directory are the most current copies of the files which the server has. Given this, we can detect a change by hashing a local file and comparing it to the hash of the file in the replicate directory (if it exists). If we detect that the hashes are different, then we may take a diff of the main file and the replicate file, and we may send this diff to the server via a `req_update` message. If the file does not exist in the replicate directory, then we would think that the file is new and we would send the full file to the server via a `req_updatefull` message. On the other hand, if a file in the replicate directory does not exist in the main sync directory, then we would think that the file has been deleted and we can notify the server via a `req_delete` message.

The main problem is how do we guarantee that the replicate directory accurately reflects the files on the server? Because the client application may be shut down for

unknown periods of time, during which another client application could be run elsewhere, it is clear that we cannot trust the contents of the replicate directory on start-up even if we had succeeded in previously guaranteeing the desired property. Thus, on start-up, the client application syncs with the server via a `req_checkall` message. In response to this, the server may send `req_requpdate`, `req_requpdatfull`, `req_update`, `req_updatefull`, and `req_notifty` messages. The client application waits and processes each of the preceding 4 types of messages until it receives a `req_notify` message. In particular:

- For a `req_requpdate` message, the client application examines the hash of the file on the server and checks if it matches the hash of the same file in the replicate drive. If it does match, then the client computes a patch with the file in the replicate drive and the file in the main drive and sends the server a `req_update` message with the patch. Otherwise, another client application must have been connected since the client application last ran and *we assume that the file that the client has is the most recent* and send the file via a `req_updatefull` message.
- For a `req_requpdatefull`, the client application copies over the current the file in the main sync drive to the replicate drive and sends the server the full file via a `req_updatefull`. Note that it is possible that the file in the main sync directory is deleted while this occurs; *we assume that a file does not get deleted while it is being synced with the server.*
- For a `req_update` message, the client application first patches the file in the replicate directory. If the hash of the file in the main sync directory matches that of the hash in the message, then the file in the main sync directory is patched as well.
- For a `req_updatefull` message, the client application first writes the file to the replicate directory, and then copies the file to the main sync directory. It is possible that the client creates a file with the same name in the main sync directory; in this case, the file in the main sync directory will be overwritten.

Though it appears that we have gotten away with the file being modified in the latter two cases, some users may be confused by the last case, where their change is overwritten. Thus, it seems appropriate that *we assume that a file is not modified while it is being synced with the server.*

We note that the server is passive in the sense that it does not send `req_update` or `req_updatefull` messages unless it is sent `req_checkall` messages. Thus, since there might be another instance of the client application running elsewhere and could be modifying the files there, we must in fact periodically send `req_checkall` messages to discover new files.

The full algorithm for the client is thus as follows: periodically perform full syncs with the server via `req_checkall` messages, and normally sync with the server using the replicate directory (via `req_update`, `req_updatefull` and `req_delete` messages as mentioned earlier).

3. Possible Improvements

There are several issues with our current implementation.

First of all, other than a simple 8 bit user identification field, there is no security within our protocol. Although currently only one set of files can be synced, this

could be easily changed. However, ensuring that one user only has access to their own files would require a revised protocol.

Local security is also an issue, as both the server and the client make system calls. If some of the commands (such as `diff`, `patch`, etc. point to different or compromised files, then these would be executed with unknown and potentially harmful effects. This can very easily be fixed by including the libraries for these commands, and calling them internally.

Secondly, we have no support for directories within the sync drive and we do not support non-text files. The former can be fixed by recursively applying our algorithms for the client and server. The latter may be fixed by choosing a better diff implementation which supports comparisons of binary files.

Third, the client implementation can be greatly improved. For example, instead of scanning the entire directory for changed files, it is possible to use the `inotify` library to watch for changes in files. This would involve, however, creating separate threads for scanning and syncing and one would have to be careful with concurrency issues. Most of the assumptions we make may be avoided by reading the entire file into memory and working with copies of the file instead of the file directly, as well as being careful with additional timestamping on the files.

Finally, the server currently can only handle one connection at a time. This does not mean one client at a time, only that clients will block the listening port until transmission is finished. This is not a big issue with small files and few clients, and could be rectified by using `select()` and threading within the server.

4. Conclusion

We learned a lot in implementing this cloud storage application, and gained newfound appreciation for existing applications such as Dropbox and Google Drive. As we noted, we have many ideas for improving our project, and we may continue developing it in the future.