File Transfer v2.0 Class Design

Friday, February 21, 2020 8:59 AM

Upload UI

Download UI

No matter what current version is, this api should be stable!

Upload Service Provider
+ registerFile(file: File)
+ unRegisterFile(file: File)

Download Service Provider
+ connectToPeer(peerId)
+ reqFileinfo(peerId, fileId)
+ startDownload(peerId, fileId)
+ pauseDownload(downloadId)
+ cancelDownload(downloadId)

V1

Not aware of anything related to peer and connections

Msg Handler

DATA

Upload Controller
+ getMainPeerId()
- listenOnConnections(fileId, remotePeerId)

Download Controller
+ getMainPeerId()
+ connectToPeer(remotePeerId)
+ reqFIleinfo(peerId, fileId)
+ startDownload(peerId, fileId)
+ pauseDownload(downloadId)
+ cancelDownload(downloadId)
- fetchCertainChunk(chunkIdx, chunkSize)

After request fileInfo, the download controller will get a list of peerId of upload workers, then after staring download, controller will generate same amount of worker peers connecting to the upload worker and send request to central handler.

Tips: data storage for data connections and peerIds should be flatten.

dataConnection will be passed on onConnection callback.
we can set a fileinfo req callback to return fileinfo.
We can also set a startDownload req callback to return a set of uploader workers' peerId.

Tips: For each data connection with the same remotePeerId, it should return same peer list.

Same group of upload workers will be bound same simple logic to get certain file block and passed to caller.

Tips: binary encode will be used.

```
// download main peer connecting to uploader main peer
// making sure, two main peers only have one dataConnection
interface UploadConnectionsEstablished {
    [remotePeerId: string]: DataConnection;
}

// there are many choices: same upload workers by ~~peer(because it's binding sending data related to file)~~/by file/by downloadId/by peer and file

// we choose the last one, distinguish by both peer and file
interface UploadWorker {
    [remotePeerId: string]: {
        [file: string]: [xxx, yyy];
    }
}

interface UploadWorkerStorage {
    [workerId: string]: {
        peerObj: Peer;
        reservedAttrN: null;
    }
}
```

```
interface DownloadConnectionsEstablished {
    [remotePeerId: string]: DataConnection;
}

interface FileStorge {
    [file: string]: {
        fileName: string;
        fileSize: number;
        uploadWorkers: {
            // support download multi times
            [downloadId: string]: {
                [uploadWorkerId: string]: [];
                downloader: FileDownloader;
                toGetChunkList: [];
            }
        };
    }
}

// status: 1 - available; 0 - busy
interface UploadWorkerStorage {
    [uploaderWorkerId: string]: {
        [downloadId: string]: {
            downloadPeerObj: Peer;
            status: 0 | 1;
            abortTimerReturnValue: 0;
        }

    }
}
```

// The downloading process is parallel, while writing process is sequential. So we need FileDownloader class.

pushCachedBlob
Every time this function triggered, will write to fileSystem via DownloadFileUtils
And the callback is triggered, the uploadWorkerStorage is updated.

if download process is not end at N seconds, prepend the chunkIdx to the toGetChunkList.

File Storage
+ addFile(file)
+ setPasswd(prev, curr)
+ getBlobByChunk(chunkIdx, chunkSize)
+ getFileInfo()
+ validPasswd(passwd)

For worker

FileDownloader
constructor(fileSize: number, cb: (worker: string) => void)
pushCachedBlob(chunkIdx)

DownloadFileUtils
+ saveBlobToFile
+ closeFile
+ cancelFile
+ queryWrittenFile

```
// 0-no-download, 1-downloading
// 2-downloaded-not-write, 3-finished
interface DownloadRecord {
    chunkList: [];
    chunkDetail: {
        [chunkIdx: number]: {
            status: 0 | 1 | 2,
            blobData: (binary);
            startTime: 111;
            endTime: 222;
            downloadWorker: 123
        }
    };
}
```