moCloud--客户端设计文档

# 历史

|  |  |  |  |
| --- | --- | --- | --- |
| **Version** | **Author** | **Date** | **Description** |
| V1.0.0 | WuJinlei | 20180226 | Create |
| V1.0.1 | Wujinlei | 20180328 | 增加下载文件(开始、停止)的流程； |
| V1.0.2 | Wujinlei | 20180409 | 修正文档，将一些实现中新增的流程，修正到文档中来； |
| V1.0.3 | Wujinlei | 20180414 | 更新文档：   1. 下载文件的流程更新为最新的实现方式； 2. 增加启动时的序列图； 3. 更改内存和线程的实际信息； |
|  |  |  |  |

# 目录

[moCloud--客户端设计文档 1](#_Toc23454)

[历史 1](#_Toc22898)

[目录 2](#_Toc18733)

[1.概述 4](#_Toc15110)

[2.应用场景 4](#_Toc27726)

[2.1.启动 4](#_Toc9775)

[2.2.注册/登录 5](#_Toc10452)

[2.3.初始化 6](#_Toc953)

[2.4.获取文件列表 7](#_Toc12158)

[2.5.下载文件 7](#_Toc2055)

[2.5.1.开始下载 7](#_Toc30546)

[2.5.2.暂停下载/停止下载 8](#_Toc22892)

[2.6.在线播放文件 8](#_Toc4891)

[2.7.上传文件 8](#_Toc13892)

[2.8.删除文件 8](#_Toc19355)

[3.序列图 9](#_Toc210)

[3.1.启动 9](#_Toc27749)

[3.1.1.密钥协商 10](#_Toc1850)

[3.2.注册/登录 10](#_Toc24705)

[3.2.1.注册 10](#_Toc20488)

[3.2.2.登录 10](#_Toc25934)

[3.3.初始化 10](#_Toc14367)

[3.4.获取文件列表 10](#_Toc3379)

[3.5.下载文件 10](#_Toc21776)

[3.6.上传文件 11](#_Toc22818)

[3.7.删除文件 11](#_Toc26397)

[4.上下文图 11](#_Toc9987)

[5.模块图 11](#_Toc16512)

[6.内存和数据 11](#_Toc25709)

[6.1.线程 11](#_Toc29606)

[6.1.1.心跳包发送线程 11](#_Toc12396)

[6.1.2.文件信息列表刷新线程 11](#_Toc26747)

[6.1.3.文件接收 12](#_Toc11290)

[6.2.内存 12](#_Toc14600)

[6.2.1.获取文件列表 12](#_Toc31059)

[6.2.2.下载文件 12](#_Toc30689)

[6.2.3.上传文件 13](#_Toc8152)

[7.类图 13](#_Toc22337)

[8.流程图 13](#_Toc5261)

[9.头文件 13](#_Toc7619)

[10.开发阶段 14](#_Toc15298)

[10.1.M1--创建模型、支持基本功能 14](#_Toc32276)

[10.2.M2--上传、下载、续传文件 14](#_Toc18082)

[10.3.M3--删除文件、在线播放文件 14](#_Toc20762)

[11. 遗留问题 15](#_Toc3390)

[11.1Client收到response，是否需要寻找mark 15](#_Toc28252)

[11.2Client发现读写server失败后的处理 15](#_Toc20944)

[12.测试 15](#_Toc31928)

[12.1.功能测试 15](#_Toc32549)

[12.2.性能测试 15](#_Toc9633)

[12.3.边界测试 15](#_Toc17235)

# 1.概述

moCloud定位为一个私有云系统，包含了稳定运行的云服务器，和可以正常使用的云客户端；

Server运行在树莓派上，client实现在手机上。两者可以通过局域网互联，是moCloud正常交互的前提，后续可以考虑将服务扩展至公网。

由于树莓派本身的硬件性能和存储size有限，因此设计方案是：

1. Server端存放一套数据文件；
2. 所有的客户端看到的都是这一套数据；
3. 客户端分两种权限：管理员和普通用户；普通客户端只能上传文件、下载文件、播放文件、获取文件列表等；管理员除此之外，还可以删除文件；

本文档，介绍moCloud的客户端的设计方式，实现流程，可以为后续参与该项目的人提供参考，也可以为后续维护提供依据。

客户端，由C语言实现的so和android上的App两部分配合完成；

So文件负责与服务器通信；App负责用户交互界面的实现；

# 2.应用场景

客户端的使用流程为：

* 启动客户端；
* 注册/登录客户端；
* 展示文件列表；
* 上传文件；下载文件；播放文件；删除文件；
* 退出登录；
* 退出客户端；

## 2.1.启动

客户端程序运行时，首先应完成启动的动作，只有完成了启动这个步骤，我们才认为具备了与server进行交互的能力，才进行后续的注册/登录、文件操作等功能；如果启动失败，客户端只能退出登录，或调整后重新登录；

启动这个动作，主要负责：与server建立连接；与server进行密钥协商，得到本轮的密钥。过程描述如下：

1. 读取配置文件，获取客户端的ip、port(包括控制端口和数据端口)：
   1. 读取配置失败，启动失败，提示用户后退出启动流程；
2. 创建控制用的socket，连接server：
   1. 创建socket失败，启动失败，提示用户后退出启动流程；
   2. 连接server失败，启动失败，提示用户后退出启动流程；
3. 密钥协商，获取本次通信使用的密钥：
   1. 密钥协商失败，启动失败，提示用户后退出启动流程；
4. 启动心跳包线程；
5. 创建数据用的socket，连接server；
   1. 创建socket失败，启动失败，提示用户后退出启动流程；
   2. 连接server失败，启动失败，提示用户后退出启动流程；

**密钥协商**，负责从server端得到本次通信所使用的加解密算法，和密钥，其流程如下：

1. 构造密钥协商的request；
2. 使用本地存储的私钥，对request加密，得到request\_cipher;
3. 发送request\_cipher到server，并等待返回值：
   1. 存在一个超时时间，在该超时时间内如果没有能够得到返回值，属于协商失败的一种情况，需要提示用户后退出启动流程；
4. 得到返回值，使用私钥对返回值进行解密；
5. 将得到的密钥信息(加解密算法、密钥长度、密钥的值等)，保存在本地内存，供本次通信使用；

**心跳包线程**，指的是客户端需要间隔N秒，向服务器发送一个心跳包，告知服务器它还在进行连接，如果超过M秒没有发送，服务器会把这个客户端的连接信息删除，客户端需要重新连接才可以与server继续通信。

其具体流程，参见6.1.章节中所述；

## 2.2.注册/登录

注册和登录严格来说是两个动作，但由于体现在一起，而且业务上来说，注册后一般会有自动的登录流程，因此放在一起。这里只描述注册流程，登录流程可据此预见到：

1. 捕获注册所需要的用户名、密码信息，并据此生成一条request(header+body，body中包含了用户名和密码)；
2. 使用本轮的密钥，对request\_header加密，得到cipherRequestHeader；
3. 发送cipherRequestHeader和**requestBody(明文)**，并等待返回值：
   1. 超时时间内得不到返回值，认为注册失败，提示用户后，退出注册流程，但不退出客户端，用户可以使用本次密钥协商的结果，继续注册；
4. 对返回值cipherResponse，使用密钥信息进行解密；
5. 分析解密后的response：
   1. 用户名已经被使用：必须更换一个用户名；
   2. 注册成功，可进行登陆操作了；

登录操作与其流程类似，不做赘述；

## 2.3.初始化

登录之后，自动执行初始化的动作，初始化主要包括两部分功能：获取文件列表并展示；继续上次未完成的上传、下载任务。

获取文件列表并展示，具体流程可以参见2.4详述的流程；

继续上次未完成的任务，例如上传、下载文件等，以上传文件为例，流程如下：

1. 查看是否存在unCompletedTask.log；
2. 解析文件，查看是否存在未完成的上传文件的任务；
3. 判断这些文件，是否依然存在于本地目录中，将其分类后，提示用户：有多少文件上次未完成，上次的时间是何时，现在有多少文件还存在、可以继续上传，上次上传了多少进度；
4. 请用户主动选择，是否继续上传，用户可以选择其中的一个或多个，可以选择上传了一部分的，或者完全没有上传过的；但是用户选择的，一定是当前本地存在的文件；
5. 如果选择了没有上传过的文件，直接执行2.1.7.的上传文件流程即可；
6. 如果选择了上传了一部分的文件，
   1. 首先向服务器发送请求，询问上次发送的文件是否还存在，是否还是该用户在该时间上传的：
   2. 等待返回值，支持超时时间，超时时间内没有返回，认为失败，请用户重新选择；
   3. 解密返回值，分析返回值：
      1. 如果依然是该用户上传的，并且size、offset等基本信息都正确，那么直接执行2.1.7的流程，继续上传文件；
      2. 如果不再是该用户上传的文件，或者文件在server端不存在了，或者offset等信息匹配失败，都需要告知用户该情况，并提示上传失败；

上传文件可以考虑有两个阶段开发，断点续传的功能，放到第二个阶段；

继续上次未完成的下载文件的流程，概述如下：

1. 启动后，查看dwldInfoFile文件，读取所有未完成的下载任务；
2. 提示给用户，当前哪些文件未下载完成，下载进度是多少了，文件总大小是多少，由用户选择继续下载哪些文件；
3. 对用户选择的继续下载的文件，发送startDwld给server，请求自offset位置起，继续下载；
   1. 如果server返回下载失败，并且是由于该文件已经不存在，那么下载将终止，并提示是否删除本地文件；
   2. 如果是其他的server下载失败，返回错误值；

每次可同时执行的下载任务，需要有数量上限，不然系统资源很难吃的消；

## 2.4.获取文件列表

文件列表展示，是moCloudClient的主界面；

有两个场景需要获取文件列表：

1. 初始化主界面时；
2. 运行过程中实时刷新主界面；

用户登录后，进入主界面，主界面就是所有的文件列表信息，因此此时需要主动发起一次查询所有类型的文件列表的request，并等待response，执行一次正常的getFileList的流程。

运行过程中，实时更新文件信息列表的主界面，采用了**异步实现**。

内置一个缓冲区，用来存放所有的文件信息列表。

每次发送心跳包给server，server如果发现其文件信息列表发生了变化，都会在heartbeat的response中置一个值，告知client需要重新获取一次列表信息。

Client发现该标记位后，主动发起一次getFileList的流程，并更新到缓冲区。在获取时，直接读取该缓冲区中的数据，返回，不再实时发送request到server，减少时间消耗，提高用户体验。

## 2.5.下载文件

下载文件，可以一次下载多个文件，但最多不能超过N个，因为服务器的能力有限，多客户端多文件下载如果不加以限制，服务器宕机就惨了。

下载文件分为两个动作：开始下载；停止下载。

开始下载也分为两个场景：下载一个新文件；继续下载一个已经存在的文件；

停止下载分为两个场景：一个文件下载完成了自动发起的停止下载(用户对此场景并不知情)；下载过程中用户通过暂停下载/停止下载的动作，主动发起的停止下载的动作；

由于需要知道下载进度，并能够进行暂停、启动等动作，因此需要本地存储。

### 2.5.1.开始下载

1. 获取开始下载的文件的offset：新文件的offset显然为0，如果是暂停下载了的文件要继续下载，那么offset就是本地存储中记录的进度文件中的信息；
2. 在CliData模块中，注册该下载任务：
   1. 启动写文件线程；
   2. 分配数据缓冲区；
   3. 注册到全局下载列表中；
3. 将相关信息(fileId，fileKey，fileLocalPath，offset等)组织为request，以ctrlRequest的形式发送给服务器，请求开始下载；
   1. 如果返回成功，step4；
   2. 如果返回失败，需要在CliData模块中，清空该下载任务的相关信息，并退出下载流程，返回下载失败；
4. CliData模块，recvData线程，开始使用数据端口，接收来自于服务器的数据(header + body + header + body + ......)；
5. 将每一个接收到的数据，写入到该文件对应的数据缓冲区中，通知writeFile线程开始进行文件写入的动作，同时将进度等信息更新到本地存储中；
6. 反复执行step4+step5，直到遇到最后一个unit为止；

这里我们约定了一个EOF的特殊response header，一旦客户端收到了该header，认为文件下载结束了，进入自动停止下载的流程，主要包括：将所有数据写入到文件；清空cliData模块中针对该下载任务的所有资源(包括线程)；清空dwldInfoFile中该下载任务的信息；

### 2.5.2.暂停下载/停止下载

1. 获取要停止的下载任务的相关信息，向moCloudServer发送停止下载的请求，并等待返回值；
2. CliData中，停止写入线程，将当前写入的文件信息更新到本地存储中后：
   1. 如果是暂停下载，保留下载的文件，step3；
   2. 如果是停止下载，判断是否需要删除已下载的文件，如果需要：删除下载的文件，删除本地存储中关于该文件的记录，step3；
3. 清空该文件对应的缓冲区，并将对应节点置为未使用，其他相关信息的设置；
4. CliData中，该fileId的相关数据，read后直接扔掉，不再写入buffer中；

## 2.6.在线播放文件

TODO

## 2.7.上传文件

TODO

## 2.8.删除文件

TODO

# 3.序列图

## 3.1.启动



图 启动序列图

### 3.1.1.密钥协商

图 密钥协商序列图

## 3.2.注册/登录

### 3.2.1.注册

### 3.2.2.登录

## 3.3.初始化

## 3.4.获取文件列表

## 3.5.下载文件

## 3.6.上传文件

## 3.7.删除文件

# 4.上下文图

# 5.模块图

# 6.内存和数据

## 6.1.线程

客户端需要支持：心跳包发送线程、文件信息列表刷新线程、文件数据接收线程；

### 6.1.1.心跳包发送线程

1. 密钥协商后启动该线程，程序退出时停止；
2. 通过返回值判断服务器是否运行正常：
   1. 能正常获取返回值：服务器正常运行；
   2. 获取不到返回值：与服务器通信中断，停止当前所有任务，等待连接；
   3. 返回值有其他意义：比如服务器上的文件列表发生变化，通过该返回值体现，告知client重新获取一次服务器文件列表，等等；

### 6.1.2.文件信息列表刷新线程

主要目的是将getFilelist的操作异步实现，原理阐述如下：

1. Client发送心跳包给server后，server的返回值告知client，文件列表(可以知道是哪个类型的文件列表)变化了；
2. 心跳包线程，发送信号量给文件信息列表刷新线程；
3. 文件信息列表刷新线程捕获到信号量后，查看client是否已经登录：
   1. 如果已经登录，继续step4；
   2. 如果未登录，不做处理，直接返回；
4. 发送getFilelist的request给server，获取给定类型的文件列表信息；
5. 将这次获取到的列表信息，设置到一个全局变量中；

至此，该线程的工作就完成了。

一旦用户需要查询文件列表的时候，并不是直接向服务器要数据，而是直接拿这个文件信息列表缓冲区中的值，这样可以减少网络通信造成的延时，提高用户体验。

### 6.1.3.文件接收

需要2个线程，一个负责接收来自于server的回复数据，称为接收线程；一个用来将数据写入到指定文件中，称为写入线程；

主要用来在**下载文件**时使用：

1. 程序启动后，启动写入线程，由于该线程以信号量作为触发机制，因此不占用系统资源；有确切的下载请求时，再启动接收线程；
2. 程序退出时停止这两个线程；
3. 接收线程接收到数据后，将数据暂存到本地内存；写入线程被唤醒后，读取本地内存的数据后，写入特定文件；

## 6.2.内存

本部分主要统计业务流程过程中，动态分配的或者size较大的内存信息。栈及小内存不在其中。

### 6.2.1.获取文件列表

获取文件列表需要一个缓冲区作为异步操作的数据交互区域。

设计为一个N个链表组成的一个数组，数组中每个元素是链表的头指针；这样做，在需要更新某一种类型的文件信息时，只需要更新数组中的一个链表就可以，无需全部链表操作。

### 6.2.2.下载文件

对于每一个下载任务来说，需要一个缓冲区，用来存放数据。

由于理论上来说，网络上收取数据的速度，远远高于硬盘上数据写入的速度。因此需要一个缓冲区，将server发过来的数据暂存到此，写入线程从该缓冲区中读取数据写入硬盘。

该缓冲区设计为一个双向链表。

为了尽可能多的减少动态分配和释放内存节点的操作，对这个双向链表做如下说明：

* 每个网络节点，存在一个标记位，标记该节点中的数据是否已经被写入硬盘，如果已经被写入了，那么这个节点就可以重新被使用了；
* 网络上收到数据，要写入缓冲区时，写入位置示意如下：



* 读取缓冲区中数据时，从head->prev开始读取，读取后，将该节点内标记位置为已经写入硬盘，再将该节点更新到head->next；

这样，只有当前链表中的数据节点都被占用了的时候，才会需要申请一个新的节点；下载任务没有完成的情况下，一个节点一旦被写入硬盘了，就会通过置标记位的方式，将其标记为可用，在网络上收到数据后，就可以重新利用到该节点。

在下载任务结束的时候，所有节点可以被释放。

### 6.2.3.上传文件

TODO

上传文件时，一次限定最多传送多少个字节的block，只需要初始化的时候，分配特定size的内存，存放这个block的size即可，由于数据部分不加密，因此这个内存的size也更好确定；

# 7.类图

# 8.流程图

# 9.头文件

# 10.开发阶段

## 10.1.M1--创建模型、支持基本功能

第一阶段，主要创建框架，保证正常的通信机制，和基本的业务模型，具体来说，主要是如下几个方面：

* 加解密模块：支持正常的加解密；
* 信息校验模块：支持需要的校验算法；
* 通信模块：client和server可以正常通信；
* 心跳包管理模块：客户端可以正常发送心跳包，服务器可以管理心跳包，两者都可以根据心跳包的正常与否，做出正确的反应；
* 基本功能之密钥协商：可以正常进行密钥协商；
* 基本功能之获取文件列表：client可以正确获取到server当前的所有文件的列表信息，并展示给用户；

1. MoCloudUtilsCrypt，moCloudUtilsCheck，打桩； OK
2. moCloudUtilsType，moCloudUtils，实现；OK
3. moCloudClient：
   1. Init: connectToServer; doKeyAgree; OK
   2. startHeartBeat : sayHiToServer; OK
   3. signUp : 注册; OK
   4. signIn : 登录；OK
   5. getFileList; OK
   6. unInit: sayByeByeToServer; freeAllResources;OK
4. moCloudServer：启动，接收client的connect请求，每一个client新建一个thread处理；
5. moCloudServer：文件管理模块；
6. moCloudServer：client管理模块，心跳包管理最重要；
7. moCloudServer：支持基本功能之密钥协商和获取文件列表；
8. 正式实现moCloudUtilsCrypt；
9. 正式实现moCloudUtilsCheck；

## 10.2.M2--上传、下载、续传文件

## 10.3.M3--删除文件、在线播放文件

# 遗留问题

## 11.1Client收到response，是否需要寻找mark

Client收到response后，是否有必要，寻找第一个可用的mark，才认为找到了response header；

现在的处理逻辑，是发送了request之后，直接等待返回值，返回值到了，直接按照response header的格式进行解析，这样是基于对tcp的信赖，我认为已经足够了，但后续如果测试发现有时候有些数据的response并不是这样满足条件的，就需要更新这部分代码：

Recv到server发来的数据后，解密，之后查找mark，找到了才认为找到了一个response header；

## 11.2Client发现读写server失败后的处理

Client发现server读写出现问题之后，应该怎么处理呢？

最好的体现，肯定是马上告知用户，然后轮询，一旦server重新上线，自动进行init操作，用户重新logIn，就可以又正常使用了。

这部分业务需要好好想想再实现。

# 12.测试

## 12.1.功能测试

## 12.2.性能测试

## 12.3.边界测试