Cloud Object Storage Synchronization: Design, Analysis, and Implementation

Fei Chen, Zhipeng Li,Changkun Jiang, Tao Xiang, and Yuanyuan Yang

摘要

不同计算终端之间的云存储同步在企业和个人用户中得到大规模使用。它使用户能够实时保持相同的数据副本，从而减轻了用户繁琐而容易出错的数据管理负担。然而，现有的云存储同步系统往往是封闭的，用户被固定在某个云服务提供商，这使得在平衡性能、成本、安全等因素时，很难从一个云服务提供商转移到另一个。为了弥补这一缺陷，本文提出了一个基于云对象存储的同步系统。具体地，我们首先通过定义一些有用的概念来说明云对象存储同步问题。然后，我们使用状态编码的思想和 push-pull 范式来提出一个云对象存储同步系统。提出的系统支持实时、多终端的云存储同步。我们还对提出的这个系统进行了原型设计。实验结果表明，本文所提出的系统有很好的应用前景。

关键词：云对象存储；同步；状态编码；push-pull 范式；系统原型

1 引言

近年来，云计算成为主流的计算基础设施[1-2]。它的特点是能够以较低的成本提供较好的计算服务。不同规模的企业也逐渐选择利用云来管理他们的数据和服务[3-7]。在云计算的各种数据管理应用中，云存储同步是有用且关键的。

通过两个应用场景可以了解云同步的重要性。对于企业用户来说，出于安全考虑，他们可能希望将数据（如静态图像）既存储在本地，又远程存储在云端，甚至将数据存储在不同的云端。云中的数据可以通过网络提供给客户访问；本地存储的数据可以作为备份。在云存储中断的情况下，可以通过将云端数据访问重定向到传统的本地访问来挽救企业的业务。对于这种应用场景，企业用户在云和本地数据存储之间同步他们的数据是至关重要的。另一个应用场景是，消费者用户也可能需要云存储同步服务。消费者用户通常有几台设备（例如，办公室电脑、家庭电脑、手机等）来处理他们的数据。他们自然希望在这些不同的设备之间同步他们的数据。云存储同步很好地满足了这一需求。如现在流行的 OneDrive[8]和 Dropbox[9]就是典型的云同步应用。

这项工作正式研究了云存储同步问题，提出了一个云存储同步系统，能够在不同的设备之间实时同步用户的数据。与现有的闭源商业产品（如 OneDrive）不同，其从使用云对象存储的终端用户的角度研究这个问题，因为对象存储是云所提供的主流存储服务。提出的系统目标是替代 OneDrive 或 Dropbox；并且是开源的，支持用户定制，避免了服务商锁定问题。用户只需要在本地安装一个同步程序就可以进行同步操作，不需要在云中安装程序。用户还可以控制自己上传的数据，避免第三方隐私泄露。所提出的系统适用于现有的主流云对象存储服务；对云没有特殊要求。它还允许用户根据他们的预算和要求选择不同的云存储服务。与现有的解决方案相比，提出的系统允许用户对自己的数据拥有更多的控制权。

2 相关工作

存储同步是计算机科学中的一个经典问题。随着新的存储平台的出现和新的技术进步（如数据结构、数学等），对它的研究一直在持续进行。现有的关于同步的研究可以分为两类：Delta 同步和完全同步。下面对这两类同步方式进行介绍。

2.1 Delta 同步

Delta 同步的主要思想是同步一个文件的变化部分，而不是整个文件内容。它通过消耗更多的计算资源来节省网络资源。这类解决方案中最典型、最重要的是 rsync 应用程序[10]。

Delta 同步可以分为两类，一类是将文件划分为固定大小的块（类似与文件系统的数据块概念）；另一类是根据文件内容确定块大小。

固定块大小。rsync 应用使用固定块大小的方法[10]。rsync 的工作原理大致如下。假设一个数据发送者想与一个数据接收者同步数据。在 rsync 的同步过程中，发送方首先向接收方发送一个目标文件的同步请求。收到请求后，接收方根据固定的块大小将目标文件分成若干块，计算每个块的校验和，然后将校验和组成一个列表。每个块的校验和包含两个哈希值。一个是弱散列值，可以非常快速地计算；另一个是强散列值，需要更多的计算消耗，但与弱散列值相比，其碰撞概率要低很多。然后，校验和列表被发送到发送者那里。当发送方收到该列表时，它计算自己的文件块的哈希值，并将其与接收方的校验和列表进行比较。当发现一个具有相同强哈希值的块时，发送方就会记录这个块的位置并跳过它。如果哈希值不一致，发送者就逐个字节地处理文件，直到找到具有相同哈希值的文件块。比较之后，发送方生成一个差异列表和相同块的位置。然后将这个列表发送给接收方。最后，接收方根据其本地文件块和收到的列表重新组织新的文件。

后来的针对固定块大小的 Delta 同步的研究都在尝试从不同的方面对 rsync 进行改进。张等人提出了 DeltaCFS 应用[11]，解决 rsync 应用需要一个个字节地比较文件内容，消耗大量的计算资源的问题。吴等人提出了 PandaSync 应用[12]，当文件大小小于某个阈值时，PandaSync 使用完全同步。否则，该应用程序使用 Delta 同步。阈值通常为 300KB-400KB，并根据系统性能进行相应的计算。

根据文件内容确定块大小。除了优化计算文件差异的过程，研究人员还研究了优化文件分块的方法。在这方面，研究人员建议将文件划分为具有可变大小的块，即块大小由文件内容决定。它通常对某个文件块利用散列算法或指纹算法进行计算，如果结果满足预先设定的条件，就可以确定一个块。因此，产生的块大小是可变的，这与 rsync 使用的固定块不同。与固定块大小的方法相比，这种方法通过增加网络资源消耗来减少计算的消耗。

QuickSync 系统使用可变数据块大小[13]。它根据网络条件选择块的大小。同时，它采用了更新延迟同步机制来同步所有的数据块，这减少了网络资源的消耗。Dsync 系统将 rsync 与根据内容确定文件块大小相结合，在计算资源和网络资源之间找到平衡[14]。它还提出了一种新的散列算法和通信协议。新的散列算法具有更快的计算速度，同时实现了相似的碰撞概率。使用新的通信协议，系统首先请求一个弱散列进行比较，如果弱散列匹配，再请求一个强散列。因此，它减少了网络通信所需的数据量。它还将几个连续的弱散列合并成一个大块，以进一步节省网络请求的数量。

2.2 完全同步

完全同步的主要思想是对于一个产生改变的文件，同步文件的全部内容，而不是改变的内容。同步整个文件的内容会消耗更多的网络资源。然而，与 Delta 同步相比，完全同步不需要计算同步双方文件之间差异，这种方法大大降低了对计算的消耗[13]。

研究人员发现，在同步小文件时，完全同步比 Delat 同步表现得更好。小文件的大小通常不超过 1MB。对于这样的文件，Delat 同步需要将文件分成若干块并计算出哈希值，这会导致了额外的网络流量，其成本超过了文件内容本身的大小[12]。因此，对于小文件来说，使用完全同步更为合适[15]。

PandaSync 系统结合了完全同步和 delta 同步[12]。它对小于 400KB 的文件采取完全同步的方式。以前的研究表明，小文件占所有文件的大多数[16-20]。大致上，80% 的日常使用的文件属于小文件[16-17]，而超过 80% 的这些文件都小于 128KB[20]。

因为不存在复杂的计算，完全同步是比较容易实现的。一些用户量巨大的公司也选择使用完全同步。谷歌的 Google Drive 和微软的 OneDrive 的同步方法就是完全同步[8,21]。因为完全同步只需要在客户端进行较小的计算，这对移动设备是友好的。本文提出的系统也采用了这种方法。

3 本文方法

3.1 本文方法概述

本文提出的云存储同步的系统架构如图 1所示。一个云存储同步系统包括两个角色：用户和云对象存储。一个用户可能有多个客户端设备；所有设备都运行云存储同步系统以保持用户数据的一致性。云对象存储可以是任何现有的商业云存储服务。用户的数据被组织成一个包含多个文件和子目录的目录，以递归的方式进行组织。它们由用户的客户端设备的本地文件系统管理。当它们被存储在云中时，所有的文件和目录都被存储为对象。

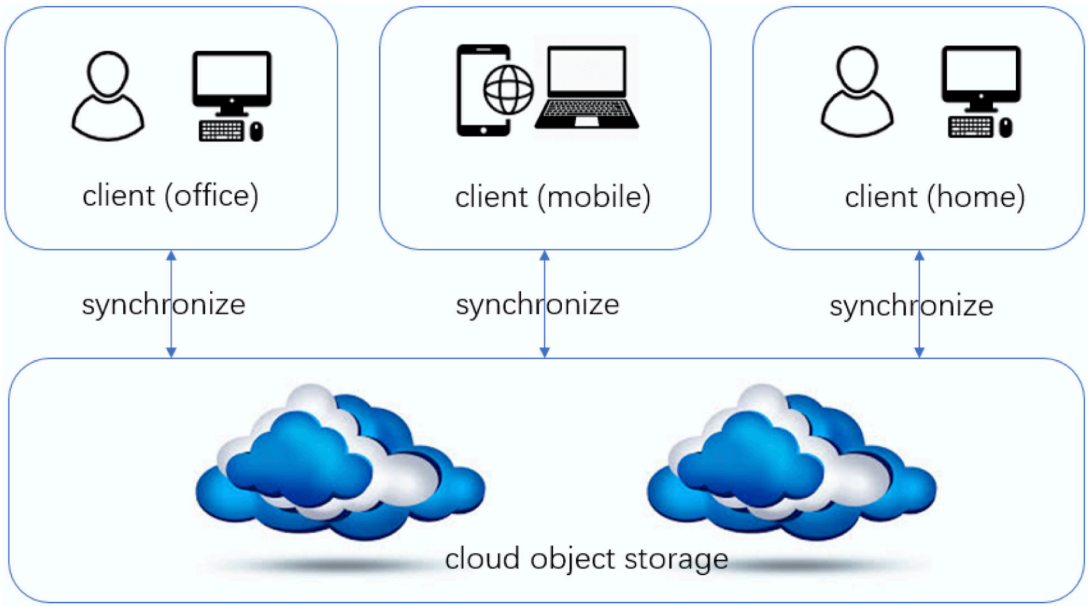

图 1: 云存储同步架构[22]

同步系统通过使用一个树形的数据结构来编码目录的状态信息，简称为状态树。该数据结构包含文件名、修改时间和哈希值作为元信息。总共使用了四个这样的状态树，其中两个用于编码本地目录的历史和当前状态，分别称为本地历史状态树和本地当前状态树；另外两个用于表示云端目录的历史

和当前状态，分别称为云端历史状态树和云端当前状态树。

当云同步系统首次运行时，本地和云端的历史状态树都初始化为空。系统会计算本地和云端的当前状态树。通过比较本地历史状态树和本地当前状态树，系统将本地存储的数据同步到云存储。我们称这个过程为 PUSH。类似的，用户也通过比较云端历史状态树和云端当前状态树，将云端的数据同步到本地。我们称这个过程为 PULL。这样就完成了一轮的同步。然后，系统利用本地和云端刚刚使用的当前状态树更新对应的历史状态树。每轮同步后，系统还将历史状态树序列化存储在本地的硬盘中，这有助于系统在一个客户端设备关闭的情况下顺利运行。

初次同步后，系统会定期同步本地目录和云目录。同步的过程与上述完全相同。在每一轮同步中，系统运行 PUSH 和 PULL 进程。同步频率可以是几秒钟也可以是几分钟，这是用户可设置的。对于实时同步，典型的同步间隔可以是 10 秒。

3.2 数据结构

本文使用树形结构来编码同步目录的状态信息。树形数据结构支持目录遍历，这对于检测文件和目录状态变化是必要的。图 3展示了状态树中的详细的数据结构。数据结构 FileStatus 编码单个文件的状态；数据结构 DirectoryStatus 是针对一个目录。DirectoryStatus 包含一个 FileStatus/DirectoryStatus 的列表作为其子节点，因为目录是由文件和子目录组成的。

```
struct FileStatus{
  string name;
  int mtime;
  string id;
}
```

```
struct DirectoryStatus{
  string name;
  int mtime;
  string id;
  list children[];
}
```

图 2: 状态树中节点的数据结构[22]

FileStatus 包含一个文件的元信息。它的作用类似于树中的叶节点。它包括一个文件名、修改时间和一个唯一的 ID。文件名可以通过本地操作系统调用或云对象存储 API 调用获得。它是一个完全路径的文件名，其优点是支持简单的重命名和删除操作。修改时间是关键信息。通过修改时间，我们可以发现一个文件是否被修改。唯一的 ID 是用来唯一地识别一个文件的。有时，两个文件有相同的内容，但有不同的名字。唯一的 ID 可以用来处理这种情况。系统使用一个哈希函数来计算唯一的 ID。由于哈希函数的抗碰撞性，这可以确保正确性。如果两个文件有相同的 ID，就意味着这两个文件的内容相同；否则，说明这两个文件的内容不同。对于 DirectoryStatus，它包含一个目录的元信息。它的作用类似于一棵树的根节点。它的元素与 FileStatus 类似，只是将 ID 设置为 NULL。图 4展示了状态树的结构。

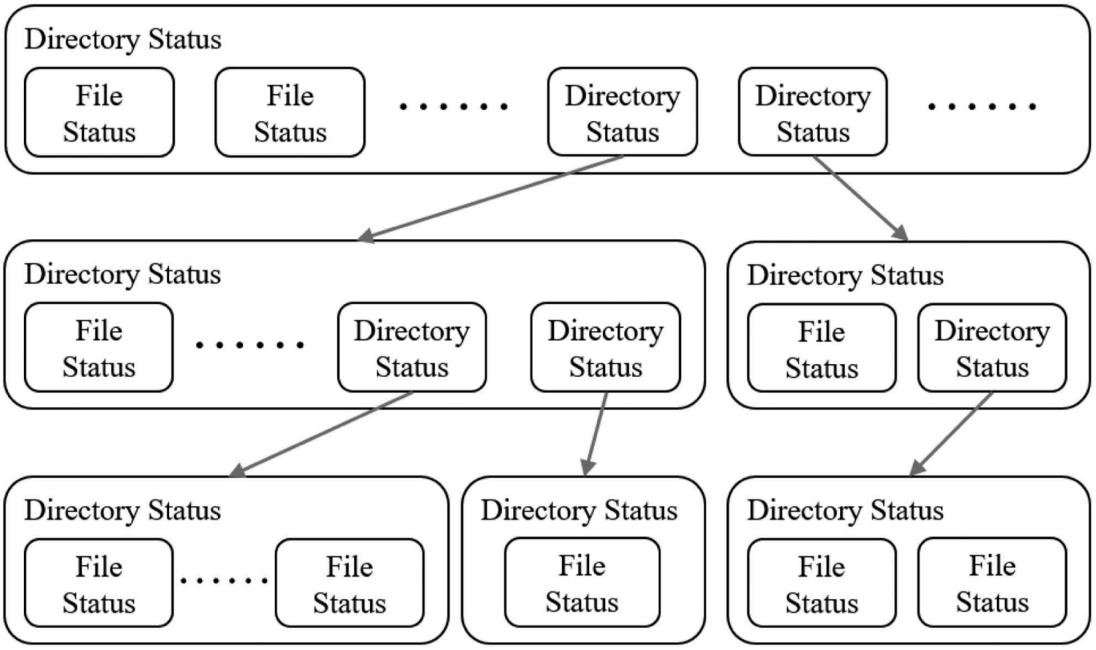

图 3: 状态树结构[22]

本文提出的云同步系统使用两个 DirectoryStatus 结构表示本地目录的历史和当前状态。同样地，使用另外两个 DirectoryStatus 结构来编码云目录的历史和当前状态。

3.3 同步算法框架

算法 1 展示了本文提出的云存储同步系统的主要框架。它主要在一个循环中运行，不断将用户的数据更新到最新状态。每个循环就是一轮同步。我们对同步框架的解释如下。对于每个子算法，我们在后面的小节进行介绍。

系统首先初始化了四棵状态树来编码本地和云端的目录信息。$\text{metaTree}_{\text{LH}}$ 和 $\text{metaTree}_{\text{LC}}$ 表示本地目录的历史和当前状态树。同样地，$\text{metaTree}_{\text{CH}}$ 和 $\text{metaTree}_{\text{CC}}$ 表示云目录的历史和当前状态树。这四个树是 DirectoryStatus 数据结构的实例。

算法 1 Cloud Storage Synchronization Framework

$\text{metaTree}_{\text{LH}}, \text{metaTree}_{\text{LC}} \leftarrow$ null
$\text{metaTree}_{\text{CH}}, \text{metaTree}_{\text{CC}} \leftarrow$ null
while true **do**
  $\text{metaTree}_{\text{LC}} \leftarrow$ BuildTree(local root path)
  PUSH($\text{metaTree}_{\text{LH}}, \text{metaTree}_{\text{LC}}$)
  $\text{metaTree}_{\text{CC}} \leftarrow$ BuildTree(cloud root path)
  PULL($\text{metaTree}_{\text{CH}}, \text{metaTree}_{\text{CC}}$)
  update and serialize $\text{metaTree}_{\text{LH}}, \text{metaTree}_{\text{CH}}$
end

然后，系统进入一轮同步。为了找出最近的状态变化，它为本地目录建立了当前状态树（即 $\text{metaTree}_{\text{LC}}$）。基于本地目录的历史和当前状态树，系统找出本地目录中的数据变化。在 PUSH 过程中，系统将这些变化同步到云端。同样，利用云端目录的历史和当前状态树，系统在 PULL 过程中将云端数据变化同步到本地目录。这样，本地和云端的数据是同步的。最后，系统将本地目录和云目录的历史状态树都更新为当前状态树，并将两个历史树存储在本地磁盘中，这在系统意外关闭的情况下很有用。这就完成了一轮的同步。然后，系统继续循环运行，进行数据同步。

3.4 构建状态树

在每一轮中，同步系统的操作都是相同的。我们选择其中的一个轮次来解释这个过程。在每一轮的开始，系统会计算状态树。

算法 2 展示了计算一个目录的当前状态的细节。首先，系统使用根目录的路径初始化一个 DirectoryStatus 数据结构。对于构建本地状态树系统使用操作系统调用。对于构建云端状态树，系统使用云对象存储的 API 调用。一个目录的 ID 被设置为 NULL；一个文件的 ID 被设置为文件内容的哈希值。然后，系统递归地遍历根目录下的所有子目录和文件。

算法 2 BuildTree

```
Input  : path that is the root directory
Output: metaTree that is the current state tree of path
metadata ← mtime, id of current directory path
metaTree ← DirectoryStatus(path, metadata)
for each file/directory f under path do
    if f is a file then
        metadata ← mtime, id of the directory file
        fs ← FileStatus(filename, metadata)
        insert fs into metaTree's children
    else
        subDir ← BuildTree(f)
        insert subDir into metaTree's children
    end
end
return metaTree
```

为了处理一个正常的文件 f，系统获得了它的元信息并构建了一个 FileStatus 对象 fs。在将 fs 插入其父目录状态树中作为一个子目录后，系统就完成了对一个文件的处理。为了处理一个子目录，系统递归地运行 BuildTree 以获得一个 DirectoryStatus 对象 subDir。系统也会将 subDir 作为子目录状态树插入其父目录状态树中。

当算法 3 完成时，系统已经遍历了根目录的所有文件和子目录。返回的 metaTree 是包含所有数据状态信息的树状数据结构。它可以在以后 PUSH 和 PULL 过程中使用，以同步用户的数据。

3.5 PUSH 算法

系统使用 BuildTree 算法状态树后，进入 PUSH 过程。PUSH 过程将本地数据变化同步到云端，如算法 3 所示。

算法 3 PUSH

```
Input : metaTree_LH, metaTree_LC
for each data d (file/directory) in metaTree_LC do
    if d is in metaTree_LH then
        if d is changed then
            update d to the cloud
        end
    else
        upload d to the cloud
    end
end
for each data d (file/directory) in metaTree_LH do
    if d is not in metaTree_LC then
        delete d in the cloud
    end
end
```

本地数据变化包括添加、修改和删除文件/目录。重命名操作可以被识别为删除和添加。为了检测这些变化，系统使用两棵状态树来编码本地目录状态变化，即本地历史状态树 $\text{metaTree}_{\text{LH}}$ 和本地当前状态树 $\text{metaTree}_{\text{LC}}$。对于添加和修改本地数据变化，变化的数据位于本地当前状态树中。为了识别它们，系统会遍历本地当前状态树中的数据，并将它们与本地历史状态树进行比较。比较之后，系统会找出这两类变化。对于删除一个文件/目录，被删除的数据并不存在于本地当前状态树中；然而，它存在于本地历史状态树中。因此，系统需要进一步遍历本地历史状态树。总之，系统使用两个遍历来完成 PUSH 过程。一个是本地当前状态树的遍历，另一个是本地历史状态树的遍历。

3.6 PULL 算法

在 PUSH 过程中系统完成了将本地设备中的数据更新同步到云端，随后系统进入了 PULL 过程。与 PUSH 过程类似，PULL 过程检测云端的数据变化并将其同步到本地设备。云端的数据变化是由其他本地设备引起的，其中包括数据的增加、修改和删除。当其他客户设备中的数据发生变化时，将使用云存储同步系统将其数据同步到云端。

算法 4 PULL

```
Input : metaTree_CH, metaTree_CC
for each data d (file/directory) in metaTree_CC do
    if d is in metaTree_CH then
        if d is changed then
            update d to the local
        end
    else
        upload d to the local
    end
end
for each data d (file/directory) in metaTree_CH do
    if d is not in metaTree_CC then
        delete d in the local
    end
end
```

PULL 操作也是通过两个遍历来识别云端的数据变化，如算法 4 所展示。在第一次遍历中，它遍历云端当前状态树 $\text{metaTree}_{\text{CC}}$，并将其与云端历状态史树 $\text{metaTree}_{\text{CH}}$ 进行比较，以找出云端的新数

据或修改过的数据。在第二次遍历中，它遍历云端历史状态树 $metaTree_{CH}$ 并将其与云端当前状态树 $metaTree_{CC}$ 进行比较，以找出云端需要删除的数据。

4 复现细节

4.1 与已有开源代码对比

这篇论文开源了它的代码[23]，是用 Python 实现的。但是，Python 是解释型语言，每次执行程序都需要进行一次编译, 因此解释型语言的程序运行效率通常较低，而且它不能脱离解释器独立运行。所以想到了用更加高效的 C++ 去复现论文提出的云同步系统，C++ 是编译型语言，其效率接近于 C 语言，在程序执行之前，有一个单独的编译过程，将程序翻译成机器语言，以后执行这个程序的时候，就无需再进行编译，直接运行可执行文件即可。总之，C++ 的执行效率是比 Python 更高的，更适合实现云同步系统这种应用级程序。后面的实验结果也说明了 C++ 的高效，相比于论文基于 Python 的实现，系统四个核心部分算法性能提升 1.5 到 4 倍左右。

在用 C++ 复现的基础上，我还对系统进行了如下完善，改进了同步系统的可用性。

1. 对同步算法进行完善，修改历史状态树更新的方法，解决因为历史状态树更新不及时，而出现的同步缺陷。

2. 修复主流文件系统不区分文件名大小写带来的同步缺陷。

4.2 复现云同步系统

复现的云同步系统采用 C++ 实现，目前对接的云端是阿里云对象存储，代码量大约三千行左右, 已经开源在 Github[24]。

UML 类图如图 4所示，CloudFileSystem 类对阿里云提供的 API 接口进行了封装，实现对云端操作的方法。

系统使用的设计模式是观察者模式，它定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。在观察者模式中，主体是通知的发布者，它发出通知时并不需要知道谁是它的观察者，可以有任意数目的观察者订阅并接收通知。使用观察者模式的原因是，通过观察者模式我们可以绑定多个不同的云端，将数据备份到不同的云端。观察者模式的实现主要在 Subject 和 Observer 这两个类，他们是互相关联的关系。Observer 类依赖于 CloudFileSystem 类，并利用本地系统调用，实现了云同步系统对本地和云端操作的方法。我们可以创建多个 Observer 对象，绑定到一个 Subject 对象上，就可以实现本地数据备份到不同云端的功能。

状态树功能的实现主要在 StateBase、FileState 和 DirectoryState 这三个类，StateBase 类是基类，包含了 FileState 和 DirectoryState 这两个类共有的属性。FileState 和 DirectoryState 类继承 StateBase 类，分别表示一个文件的状态和一个目录的状态。

Synchronize 类是同步系统算法的实现类，通过观察其方法就可以得知。Synchronize 类依赖于 StateBase 类，使用了多态技术。Synchronize 类还依赖于 Subject 和 Observer 类，用于实现观察者模式。

Logger 类用于实现系统的日志输出，能够帮助我们了解并监控系统状态，在发生错误或者接近某种危险状态时能及时提醒我们处理，同时在系统产生问题，能够帮助我们快速定位、诊断问题。

图 4: 复现的云同步系统的 UML 类图

4.3 修复同步缺陷

在用 C++ 复现的基础上，我还修复了论文提出的同步系统所存在的一些缺陷，改进了同步系统的可用性。主要有以下两点，后面将进行详细介绍。

1. 对同步算法进行完善，修改历史状态树更新的方法，解决因为历史状态树更新不及时，而出现的同步缺陷。

2. 修复主流文件系统不区分文件名大小写带来的同步缺陷。

第一个同步缺陷是历史状态树的更新不及时。论文中同步算法的历史状态树更新是在每轮同步的最后，将当前状态树直接赋值给对应的历史状态树，这样会因为历史树更新不及时，而出现同步缺陷。我们可以假设这样一个场景，有两个设备同时开启云同步系统，分别为 host1 和 host2，同步的流程可以参考算法 1。在 host1 的第一轮同步的前，我们往 host1 本地的同步文件夹添加一个文件 file1，在 PUSH 操作的时候，这个文件会被同步到云端，并且 host2 在其第一轮同步从云端将 file1 文件拉取到了本地；在 host1 第二轮同步开始前，用户把 file1 文件从 host1 本地删掉，正常的同步是在第二轮同步，系统会把云端和 host2 的 file1 文件删掉，但是此时的 host2 进行 push 时，本地历史状态树是没有 file1 记录，但其本地当前状态树有 file1 记录，系统会从 host2 把 file1 文件再次同步到云端，此时 host1 在第二轮同步的 pull 操作又把刚刚删掉的文件重新同步到了本地，这显然违背了 host1 删除文件的初衷。上述同步缺陷只是因为历史树更新不及时所导致的缺陷之一。

所以我将历史状态树的更新更改为在每次同步操作（上传、下载、更新和删除等）后去修改历史状态树中对应的值，这样的话历史状态树的更新是足够及时的。以为刚刚的那个场景为例，如果 host2 文件夹的历史树更新是及时的，那么 host2 在第二轮同步 PUSH 的时候，本地历史状态树应该是有 file1 记录的，并且本地当前状态树有 file1 记录，不会把 file1 文件再次传回云端，这样也就不存在上述举例的同步缺陷了。

对同步算法历史状态树更新方法完善后的算法如算法 5 所示，在每轮同步的最后系统只进行序列化历史状态树到本地而不需要更新历史状态树，历史状态树的更新是在 PUSH 和 PULL 操作中每次同步操作（上传、下载、更新和删除等）后去修改历史状态树中对应的值。

算法 5 Cloud Storage Synchronization Framework

```
metaTree_LH, metaTree_LC ← null
metaTree_CH, metaTree_CC ← null
while true do
    metaTree_LC ← BuildTree(local root path)
    PUSH(metaTree_LH, metaTree_LC)
    metaTree_CC ← BuildTree(cloud root path)
    PULL(metaTree_CH, metaTree_CC)
    serialize metaTree_LH, metaTree_CH
end
```

在 PUSH 操作中更新本地历史状态树和云端历史状态树，如算法 6 所示，将其与算法 3 进行对比，可以清楚的看到完善后的历史状态树是如何更新的。

算法 6 PUSH

```
Input : metaTree_LH, metaTree_LC
for each data d (file/directory) in metaTree_LC do
    if d is in metaTree_LH then
        if d is changed then
            update d to the cloud
            update d in metaTree_LH
            update d in metaTree_CH
        end
    else
        upload d to the cloud
        insert d in metaTree_LH
        insert d inmetaTree_CH
    end
end
for each data d (file/directory) in metaTree_LH do
    if d is not in metaTree_LC then
        delete d in the cloud
        remove d in metaTree_LH
        remove d in metaTree_CH
    end
end
```

在 PULL 操作中更新本地历史状态树和云端历史状态树，如算法 7 所示，将其与算法 4 进行对比，可以清楚的看到完善后的历史状态树是如何更新的。

算法 7 PULL

```
Input  : metaTree_CH, metaTree_CC
for each data d (file/directory) in metaTree_CC do
    if d is in metaTree_CH then
        if d is changed then
            update d to the local
              update d in metaTree_CH
              update d in metaTree_LHend
        else
            upload d to the local
              insert d in metaTree_CH
              insert d inmetaTree_LHend
        end
        for each data d (file/directory) in metaTree_CH do
            if d is not in metaTree_CC then
                delete d in the local
                  remove d in metaTree_CH
                  remove d in metaTree_LHend
            end
```

第二个同步缺陷是因本地操作系统不区分文件名大小写，而云端区分文件名大小导致的。本地的操作系统，如 Windows、Linux 和 MacOS 等都是不区分文件名大小写的，即将只有文件名大小写不一样的文件当作同一文件，如本地的 File1.txt 和 file1.txt 是同一文件。而云端是区分文件名大小写的，如云端的 File2.txt 和 file2.txt 是不同的文件。因为本地和云端对文件名大小写的处理是不一致的，会存在同步缺陷。解决办法是，当在状态树中查找某个文件或目录的状态时，如出现了仅文件名大小写不一致的情况，修改状态树中的对应的记录并把对应的云端或本地文件进行重命名。

4.4 实验环境搭建

我使用 C++17 对云同步系统进行了复现并完善，使用阿里巴巴云对象存储服务作为云端。系统使用 SHA-256 作为我们的加密哈希函数来计算一个文件的唯一 ID。对于阿里巴巴云对象存储服务，将存储的地理区域设置为” 深圳”；我们没有使用服务提供商的加速服务，如 CDN。为了运行客户端程序，我们采用了一台运行 MacOS 操作系统的台式电脑，配备英特尔 i7 处理器、16G 内存和 512G 的 SSD 磁盘。这台台式电脑位于深圳大学，有 10Mbps 的外网连接。系统使用 cereal 库将历史状态树序列化到磁盘中，使用 C++ 的内置的 clock() 计时函数来测量算法运行时间。

4.5 界面分析与使用说明

对程序编译后，我们可以得到一个名为 cloudsync 的可执行文件。首先需要修改 configuration.json 配置文件，里面存放了阿里云的访问密钥、本地同步文件夹和云端同步文件夹等配置信息。然后通过./cloudsync configuration.json 命令就可以运行云同步系统了。如图 5所示，系统在终端还会输出运行时的日志。

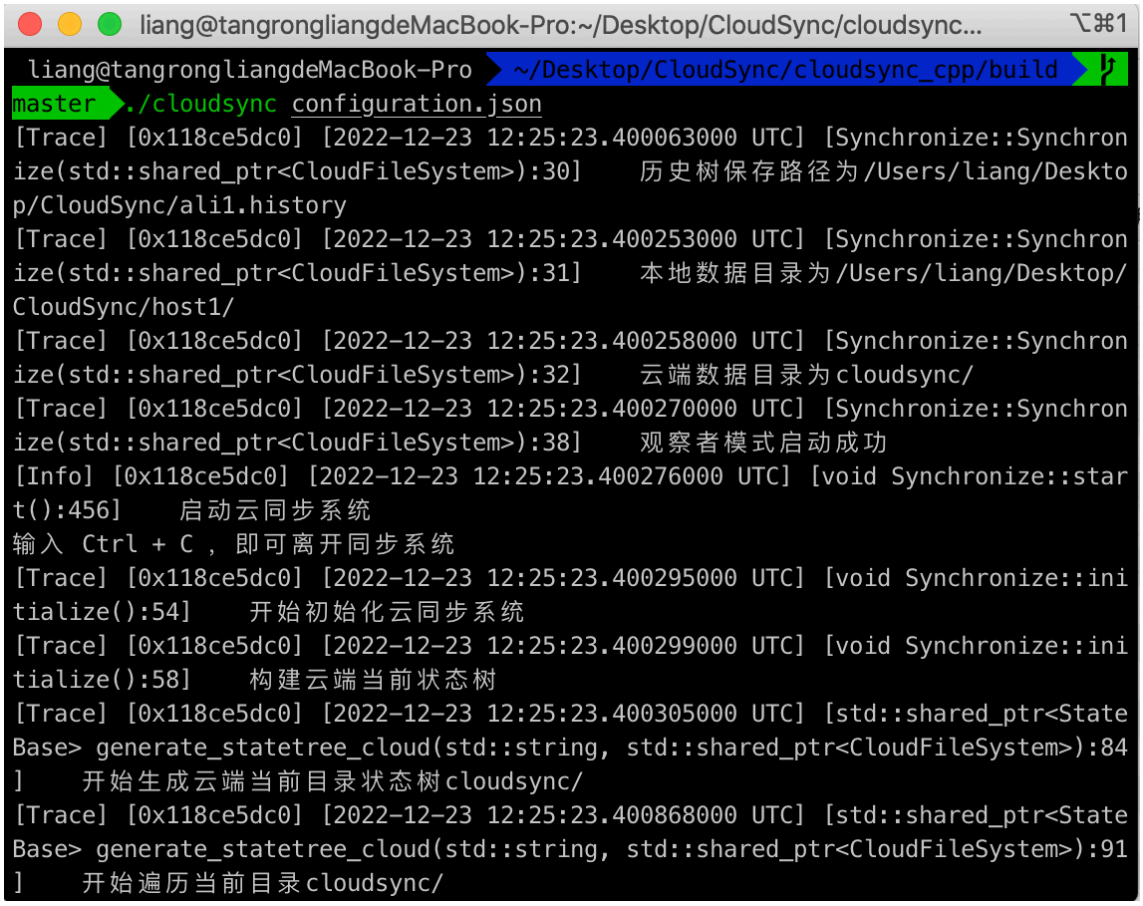

图 5: 系统使用方法

4.6 创新点

本项目的复现工作，采用 C++ 实现了云数据同步系统，支持实时、多终端、用户可定制的云存储服务同步。相比于原论文基于 Python 的实现，系统核心部分算法性能提升 1.5 到 4 倍左右，并修复了历史状态树更新不及时和主流文件系统不区分文件大小写带来的同步缺陷，同时改进了同步系统的可用性。

5 实验结果分析

为了说明用 C++ 复现的云同步系统，相比于论文用 Python 实现的系统更加高效。在保证 C++ 和 Python 实现系统的算法一致情况下，我们对其性能指标进行评估对比，主要考虑计算成本。测量了构建本地和云端当前状态树、PUSH 和 PULL 操作的时间，这些都是同步算法的核心步骤。分别进行 10 次测试，然后取平均结果。实验数据为日常使用的文件，约 3000 个文件、共 106MB 大小。

实验结果如表 1 所示，可以看到采用 C++ 复现的方法比论文中用 Python 实现的方法更加高效，构建本地树快 3 倍左右，构建云端树快 4 倍左右，PUSH 操作快 1.5 倍左右，PULL 操作快 4 倍左右。故复现中用 C++ 实现的方法，能够让论文中提出的云同步系统真正达到应用级别。

表 1: 实验结果

系统	构建本地当前状态树	构建云端当前状态树	PUSH	PULL
论文	0.36s	109.54s	89.55s	160.92s
复现	0.12s	28.71s	62.03s	42.91s

6 总结与展望

云存储服务使用户能够实时维护同一份数据副本，减轻了用户繁琐的数据管理负担，在企业和个人用户中得到了大规模应用。然而，现有的云存储服务往往是封闭且不够高效的，所以该论文提出了一个开源的、实时高效的云同步系统。本项目的复现工作，采用 C++ 实现了云数据同步系统，支持实时、多终端、用户可定制的云存储服务同步。相比于原论文基于 Python 的实现，系统核心部分算法性能提升 1.5 到 4 倍左右，并修复了历史状态树更新不及时和主流文件系统不区分文件名大小写带来的同步缺陷，同时改进了同步系统的可用性。

然而，当前的同步系统采用的是完全同步的方法，只要内容改变了，会传输整个文件，对大文件不友好，所以进一步工作可以考虑结合 delta 同步的方法来处理大文件。

参考文献

[1] From cloud first to cloud smart[J/OL]., 2021. https://cloud.cio.gov/strategy/.

[2] CHEN Y, LIN L, LI B, et al. Silhouette: Efficient cloud configuration exploration for large-scale analytics[J]. IEEE Transactions on Parallel and Distributed Systems, 2021, 32(8): 2049-2061.

[3] Mcafee. Cloud adoption and risk report 2019[J/OL]., 2020. https://cloudsecurity.mcafee.com/cloud/en-us/forms/white-papers/wp-cloud-adoption-risk-report-2019-banner-cloud-mfe.html.

[4] JIANG T, MENG W, YUAN X, et al. ReliableBox: Secure and Verifiable Cloud Storage With Location-Aware Backup[J]. IEEE Transactions on Parallel and Distributed Systems, 2021, 32(12): 2996-3010.

[5] WANG J, HAN D, YIN J, et al. ODDS: Optimizing Data-Locality Access for Scientific Data Analysis [J]. IEEE Transactions on Cloud Computing, 2017, 8(1): 220-231.

[6] LI L, LAZOS L. Proofs of Physical Reliability for Cloud Storage Systems[J]. IEEE Transactions on Parallel and Distributed Systems, 2019, 31(5): 1048-1065.

[7] CHEN F, MENG F, XIANG T, et al. Towards usable cloud storage auditing[J]. IEEE Transactions on Parallel and Distributed Systems, 2020, 31(11): 2605-2617.

[8] Microsoft onedrive[J/OL]., 2021. https://www.microsoft.com/en-us/microsoft-365/onedrive/online-cloudstorage.

[9] Dropbox[J/OL]., 2021. https://www.dropbox.com/.

[10] TRIDGELL A, MACKERRAS P. The Rsync algorithm[J/OL]. The Australian National University, 1996. https://openresearch-repository.anu.edu.au/bitstream/1885/40765/3/TR-CS-96%E2%80%9305.pdf.

[11] ZHANG Q, LI Z, YANG Z, et al. DeltaCFS: Boosting delta sync for cloud storage services by learning from NFS[C]//2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS). 2017: 264-275.

[12] WU S, LIU L, JIANG H, et al. Pandasync: Network and workload aware hybrid cloud sync optimization [C]//2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS). 2019: 282-292.

[13] CUI Y, LAI Z, WANG X, et al. QuickSync: Improving synchronization efficiency for mobile cloud storage services[C]//Proceedings of the 21st Annual International Conference on Mobile Computing and Networking. 2015: 592-603.

[14] HE Y, XIANG L, XIA W, et al. Dsync: a lightweight delta synchronization approach for cloud storage services[C]//36th Symposium on Mass Storage Systems and Technologies (MSST'20). 2020.

[15] DRAGO I, BOCCHI E, MELLIA M, et al. Benchmarking personal cloud storage[C]//Proceedings of the 2013 conference on Internet measurement conference. 2013: 205-212.

[16] PINHEIRO E, WEBER W D, BARROSO L A. Failure trends in a large disk drive population[J]., 2007.

[17] MEYER D T, BOLOSKY W J. A study of practical deduplication[J]. ACM Transactions on Storage (ToS), 2012, 7(4): 1-20.

[18] LI Z, JIN C, XU T, et al. Towards network-level efficiency for cloud storage services[C]//Proceedings of the 2014 Conference on Internet Measurement Conference. 2014: 115-128.

[19] TRAEGER A, ZADOK E, JOUKOV N, et al. A nine year study of file system and storage benchmarking [J]. ACM Transactions on Storage (TOS), 2008, 4(2): 1-56.

[20] ZHANG S, CATANESE H, WANG A A I. The Composite-file File System: Decoupling the {One-to-One} Mapping of Files and Metadata for Better Performance[C]//14th USENIX Conference on File and Storage Technologies (FAST 16). 2016: 15-22.

[21] Google. Cloud storage for work and home - Google drive.[J/OL]., 2022. https://www.google.%20com/drive/.

[22] CHEN F, LI Z, JIANG C, et al. Cloud Object Storage Synchronization: Design, Analysis, and Implementation[J]. IEEE Transactions on Parallel and Distributed Systems, 2022, 33(12): 4295-4310.

[23] CloudSync[J/OL]., 2022. https://github.com/szu-security-group/CloudSync.

[24] CloudSync-cpp[J/OL]., 2022. https://github.com/rongliangtang/CloudSync-cpp.