华中科技大学

课程报告

课程名称：高级分布式系统

专业班级：

学　　号：

姓　　名：黄浩岩

授课老师：金海、石宣化

报告日期：2023 年 1 月 31 日

计算机科学与技术学院

目 录

实验一：Paxos 算法

1 实验目的

Paxos 算法是莱斯利·兰伯特于 1990 年提出的一种基于消息传递的一致性算法。这个算法被认为是类似算法中最有效的。目前在 Google 的 Chubby、MegaStore、 Spanner 等系统中得到了应用，另外，ZooKeeper 也使用了 Paxos 算法。

本次实训考察学生对 Paxos 算法思想的理解，并完成相关程序设计。

2 实验内容

根据 Paxos 算法流程完成相关核心成员函数设计。根据提示，在编辑器中的 begin-end 间补充代码，根据 Paxos 算法流程完成 Proposer.cpp 和 Acceptor.cpp 中 Proposer 和 Acceptor 类的核心成员函数设计。

实验的相关概念知识：

- 角色：

  在 Paxos 算法中有以下几个角色:

  - ✧ 提议者(proposer): 进行提议的角色；
  - ✧ 批准者(acceptor): 通过提议的角色；
  - ✧ 学习者(learner): 感知(learn)被选中的提议。

  在具体的实现中，一个进程可能同时充当多种角色。比如一个进程可能既是 Proposer 又是 Acceptor 又是 Learner。

- 提案：

  最终要达成一致的 value 在提案里。

  Proposer 可以提出（propose）提案；Acceptor 可以接受（accept）提案；如果某个提案被选定（chosen），那么该提案里的 value 就被选定了。

- 选定提案：

Proposer：只要 Proposer 发的提案被 Acceptor 接受，Proposer 就认为该提案里的 value 被选定了。

Acceptor：只要 Acceptor 接受了某个提案，Acceptor 就认为该提案里的 value 被选定了。

Learner：Acceptor 告诉 Learner 哪个 value 被选定，Learner 就认为那个 value 被选定。

3 实验方法

本实验模拟了 Paxos 算法达成一致的过程。实验设计了 5 个 Proposer，11 个 Acceptor 准备进行 Paxos，每个 Proposer 独立线程，Acceptor 不需要线程，Proposer 编号从 0 – 10，编号为 i 的 Proposer 初始提议编号和提议值是（i+1, i+1）。Proposer 每个提议都需要得到半数以上即 6 个以上的 Acceptor 的同意，否则就要增大自己的提议编号重新发起提议。Proposer 每次重新提议会按 Proposer 的数量增加提议编号，即每次增加 5。Proposer 被批准后结束线程，其它线程继续投票最终，全部批准相同的值，达成一致。Paxos 算法包括两个阶段：Propose（提议）阶段和 Accept（批准）阶段。

3.1 Propose 阶段

Paxos 算法的 Propose 阶段，Proposer 发起提议并向 Acceptor 发出请求，Acceptor 收到请求后进行承诺并做出应答。

3.1.1 Acceptor

在实验中模拟的 Propose 阶段，Proposer 拿到提议后会向 11 个 Acceptor 发送消息，Acceptor 收到消息后，通过方法 Acceptor::Propose()处理消息。

具体而言，在 Propose 阶段，Acceptor 通过比较收到的提议消息的编号 serialNum 来判断是否通过当前提议。

（1）若当前提议的编号为 0，即 0 == serialNum，则为非正常编号（实验中提议编号从 1 开始），因此该提议不通过，返回 false。

（2）若 Acceptor 记录接收过的提议的最大编号比当前提议的编号大，即 m_maxSerialNum > serialNum，由于 Acceptor 承诺不再接受比收到的 Propose 请求编号小于等于的提议，因此该提议不通过，返回 false。

（3）通过上述条件过滤，则 Acceptor 接受该提议。然后接下来执行两步操作：

①首先更新记录的接受过的最大编号， 即 m _maxSerialNum = serialNum，从而使得 Acceptor 能够承诺不再接受比该请求编号小的提议。

②若已经在 Accept 阶段接受过提议，即 lastAcceptValue = m_lastAcceptValue，则将最后一次接受的提议返回给 Proposer，以便与 Accept 阶段接受的提议保持一致。

具体代码如下所示：

```
bool Acceptor::Propose(unsigned int serialNum, PROPOSAL
&lastAcceptValue)
{
    if ( 0 == serialNum ) return false;
    //提议不通过
    if ( m_maxSerialNum > serialNum ) return false;
    //接受提议
    //请完善下面逻辑

    /**********Begin**********/
    m_maxSerialNum = serialNum;
    lastAcceptValue = m_lastAcceptValue;
    /**********End**********/

    return true;
}
```

3.1.2 Proposer

在实验中模拟的 Propose 阶段，提议由 Acceptor 处理后再发送给 Proposer，Proposer 通过 Proposer::Proposed()方法处理 Propose 回应并判断是否进行 Accept 阶段。

（1） Proposer 首先检查自己的完成拉票的标志，即 if(m_proposeFinished == true)，则可能是由于网络延迟等问题收到的 Propose 阶段迟到的回应，因此直接忽略消息，返回 true 进行 Accept 阶段。

（2）若 Acceptor 拒绝了提议，即 if(!ok)，则增加拒绝数 m_refuseCount++。同时继续判断，若已有半数的 Acceptor 拒绝了该提议，即 if (m_refuseCount > m_acceptorCount / 2)，则无需等待其他 Acceptor 的投票结果，根据多数原则则表明该提议被拒绝了，需要重新开始 Propose 阶段。因此增大提议的编号，即 m_value.serialNum += m_proposerCount，然后使用 StartPropose(m_value)方法重置状态，并返回 false，重新进行 Propose 阶段。若该提议的拒绝数未到半数，则返回 true 继续接受其他 Acceptor 的消息。

（3）若 Acceptor 接受了该提议，首先增加接受数 m_okCount++。若之前已经有提议被接受了，即存在返回的推荐提议，if (lastAcceptValue.serialNum > m_maxAcceptedSerialNum)，则需要将提议修改成 Acceptor 返回的已被接受的提议：需要更新 Proposer 的最大接受的提议编号 m_maxAcceptedSerialNum =

lastAcceptValue.serailNum，以及更新预备提议 m_value.value = lastAcceptValue.value。

（4）若 Proposer 自己的提议被半数的 Acceptor 接受，即 if (m_okCount > m_acceptCount / 2)，则根据多数原则该提议被接受，则此时表明 Propose 阶段完成，准备进入 Accept 阶段，设置 m_proposeFinished = ture（同时清零 m_okCount）。

具体代码如下所示：

```
bool Proposer::Proposed(bool ok, PROPOSAL &lastAcceptValue)
{
    if ( m_proposeFinished ) return true;//可能是一阶段迟到的回应，直接忽略消息

    if ( !ok )
    {
        m_refuseCount++;
        //已有半数拒绝，不需要等待其它 acceptor 投票了，重新开始 Propose 阶段
        //使用 StartPropose(m_value)重置状态

        //请完善下面逻辑
        /**********Begin**********/
        if (m_refuseCount > m_acceptorCount / 2)
        {
            m_value.serialNum += m_proposerCount;
            StartPropose(m_value);
            return false;
        }
        /**********End**********/

        //拒绝数不到一半
        return true;
    }

    m_okCount++;
    /*
        没有必要检查分支：serialNum 为 null
        因为 serialNum>m_maxAcceptedSerialNum，与 serialNum 非 0 互为必要条件
    */
    //如果已经有提议被接受，修改成已被接受的提议
    //请完善下面逻辑
    /**********Begin**********/
    if (lastAcceptValue.serialNum > m_maxAcceptedSerialNum)
```

```
    {
        m_maxAcceptedSerialNum = lastAcceptValue.serialNum;
        m_value.value = lastAcceptValue.value;
    }
    /**********End**********/

    //如果自己的提议被接受
    if ( m_okCount > m_acceptorCount / 2 )
    {
        m_okCount = 0;
        m_proposeFinished = true;
    }
    return true;
}
```

3.2 Accept 阶段

Paxos 算法的 Accept 阶段，Proposer 选择提议并向接受该提议的多数 Acceptor 发送 Accept 请求，Acceptor 收到请求后检查提议是否未被自己的“两个承诺”并持久化当前提议后应答 Proposer，形成决议。

3.2.1 Acceptor

在实验中模拟的 Accept 阶段，Proposer 在 Propose 阶段会确定 1 个预备提议，然后会发送向接受该提议的多数 Acceptor 发送该提议的 Accept 请求，接受该提议的 Acceptor 收到请求消息后，通过方法 Acceptor::Accept()方法处理消息。

具体而言，在 Accept 阶段，Acceptor 同样是通过比较收到的提议消息的编号 serialNum 来判断是否通过当前提议。

（1）若当前提议的编号为 0，即 0 == value.serialNum，则为非正常编号（实验中提议编号从 1 开始），因此该提议不通过，返回 false。

（2）若 Acceptor 在该提议之后又应答了其他提议，即记录接收过的提议的最大编号比当前提议的编号大，m_maxSerialNum > value.serialNum，由于 Acceptor 承诺不再接受比收到的 Accept 请求编号更小的提议，因此拒绝该提议，返回 false。

（3）通过上述条件过滤，则 Acceptor 批准该提议通过，更新接受的提议 m_lastAcceptValue = value。

具体代码如下：

```
bool Acceptor::Accept(PROPOSAL &value)
{
```

```
    if ( 0 == value.serialNum ) return false;
    //Acceptor 又重新答应了其他提议
   //请完善下面逻辑
    /**********Begin**********/
    if (m_maxSerialNum > value.serialNum) return false;
    /**********End**********/


    //批准提议通过
    //请完善下面逻辑
    /**********Begin**********/
    m_lastAcceptValue = value;
    /**********End**********/

    return true;
}
```

3.2.2 Proposer

在实验中模拟的 Accept 阶段，提议由 Acceptor 处理后再发送给 Proposer，Proposer 通过 Proposer::Accepted()方法处理 Accept 请求的回应来判断是否得到了决议。

（1） Proposer 首先检查自己的是否还未完成拉票，即 if(m_proposeFinished == false)，如果是，则可能是由于网络延迟等问题收到的上一次 Accept 阶段迟到的回应，因此直接忽略消息。

（2）若 Acceptor 拒绝了该提议的 Accept 请求，即 if(!ok)，则增加拒绝数 m_refuseCount++。同时继续判断，若已有半数的 Acceptor 拒绝了该提议，即 if (m_refuseCount > m_acceptorCount / 2)，则无需等待其他 Acceptor 的投票结果，根据多数原则则表明该提议的请求被拒绝了，需要重新开始 Propose 阶段。因此增大提议的编号，即 m_value.serialNum += m_proposerCount，然后使用 StartPropose(m_value)方法重置状态，并返回 false，重新进行 Propose 阶段。若该提议的拒绝数未到半数，则返回 true 继续接受其他 Acceptor 的消息。这一部分和 Proposer::Propose()方法是一致的。

（3）若 Acceptor 接受了该提议，首先增加接受数 m_okCount++。若 Proposer 自己提议的 Accept 请求被半数的 Acceptor 接受，即 if (m_okCount > m_acceptCount / 2)，则根据多数原则该提议被接受，成为决议，表明 Accept 阶段完成，达成共识，此时设置 m_isAgree = ture。

具体代码如下所示：

```
bool Proposer::Accepted(bool ok)
{
```

```
    if ( !m_proposeFinished ) return true;//可能是上次第二阶段迟到的回
应，直接忽略消息

    if ( !ok )
    {
        m_refuseCount++;
        //已有半数拒绝，不需要等待其它 acceptor 投票了，重新开始 Propose 阶段
        //使用 StartPropose(m_value)重置状态
        //请完善下面逻辑
        /**********Begin**********/
        if (m_refuseCount > m_acceptorCount / 2)
        {
            m_value.serialNum += m_proposerCount;
            StartPropose(m_value);
            return false;
        }
        /**********End**********/

        return true;
    }

    m_okCount++;
    if ( m_okCount > m_acceptorCount / 2 ) m_isAgree = true;

    return true;
}
```

3.3 交互过程

实验中，通过 Proposer()函数模拟了 Paxos 算法的整个流程，流程主要即上文描述的 Propose 阶段和 Accept 阶段。

Propose 阶段发起提议：主要分为三个部分，期间两次通信交互：

（1） 首先 Proposer 创建一个编号为 S，值为 V 的提议，并发送给所有 Acceptor。

（2） 当 Acceptor 接收到提议后使用 Acceptor::Propose()方法处理提议，根据编号确定是否通过提议，然后回应该发起提议的 Proposer。

（3） Proposer 收到 Acceptor 回应后使用 Proposer::Proposed()方法处理消息，根据多数原则选择进入 Accept 阶段或者重新开始 Propose 阶段。

进入 Accept 阶段后，同样主要分为三个部分，期间两次通信交互：

（1） Proposer 确定进入 Accept 阶段后，会将确定的预备提议的 Accept 请求发送给同意该提议的多数 Acceptor。

（2） 当 Acceptor 接收到提议的 Accept 请求后，使用 Acceptor::Accept()方法同样根据提议编号确定是否不违反承诺可以被接受，然后回应 Proposer。

（3） 最后 Proposer 根据 Acceptor 的回应，使用 Proposer::Accept()方法依据多数原则确定最终的决议。

如图 1 所示，为 Paxos 算法的交互时序图。

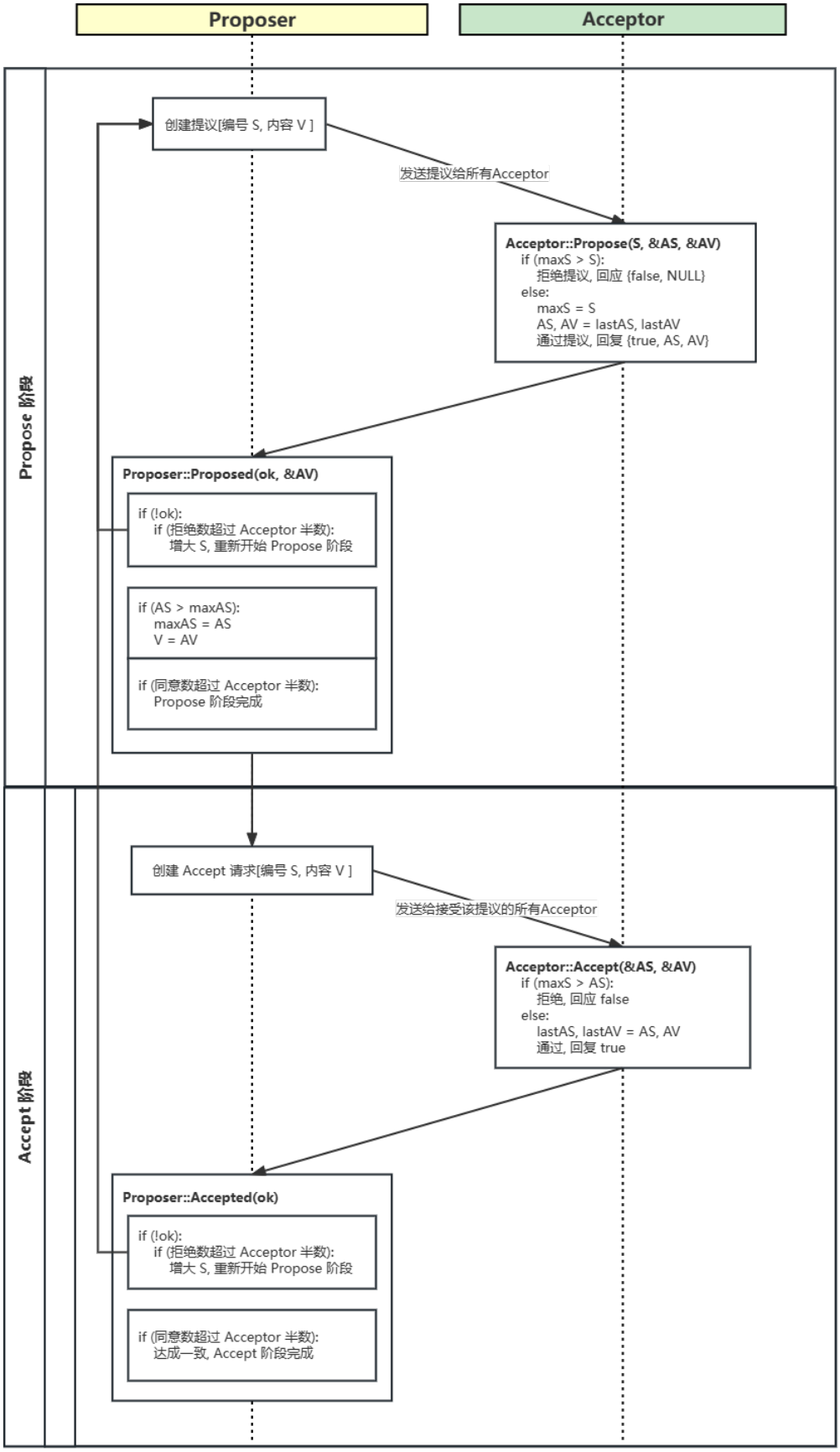

图 1 Paxos 算法时序图

4 实验结果

实验在实训平台上进行，如图 2 所示，提议最终达成一致，实际输出与预期输出一致。

图 2 Paxos 算法实训平台测试结果

通过在本地 Linux 环境部署代码并运行，可以得到更加完整的实验输出结果，如图 3 所示，可以看到 Paxos 算法开始后，5 个线程的 Proposer 的提议陆续被批准，最终提议达成一致，算法完成。

```
2023-01-20 07:44:01 Tid:87 [Info] 5个Proposer, 11个Acceptor准备进行Paxos
每个Proposer独立线程，Acceptor不需要线程
Proposer编号从0-10,编号为i的Proposer初始提议编号和提议值是（i+1, i+1）
Proposer每次重新提议会将提议编号增加5
Proposer被批准后结束线程,其它线程继续投票最终，全部批准相同的值，达成一致。

2023-01-20 07:44:01 Tid:87 [Info] Paxos开始

2023-01-20 07:44:04 Tid:92 [Info] Proposer4号的提议被批准,用时3748MS:最终提议 = [编号:5, 提议:5]

2023-01-20 07:44:28 Tid:88 [Info] Proposer0号的提议被批准,用时27411MS:最终提议 = [编号:21, 提议:5]

2023-01-20 07:44:33 Tid:91 [Info] Proposer3号的提议被批准,用时32224MS:最终提议 = [编号:24, 提议:5]

2023-01-20 07:44:37 Tid:90 [Info] Proposer2号的提议被批准,用时36913MS:最终提议 = [编号:28, 提议:5]

2023-01-20 07:44:42 Tid:89 [Info] Proposer1号的提议被批准,用时41126MS:最终提议 = [编号:32, 提议:5]

2023-01-20 07:44:42 Tid:89 [Info] Paxos完成，用时41127MS，最终通过提议值为：5
```

图 3 Paxos 算法完整输出结果

实验二：写穿算法（Write Through）

1 实验目的

写穿算法（Write Through）是分布式文件系统中保证缓存一致性的有效算法。本次实训考察学生该算法的理解，并完成相关程序设计。

2 实验内容

补充程序实现写穿算法。根据 cache.py 中的提示，完成 cache 类的读和写程序，即编写完成读函数和写函数。

实验的相关概念知识：

- 分布式文件系统：

从用户的使用角度来看，分布式文件系统是一个标准的文件系统，提供了一系列 API，由此进行文件或目录的创建、移动、删除，以及对文件的读写等操作

从内部实现角度来看，分布式文件系统还要通过网络管理存储在多个节点上的文件和目录。并且，同一文件不只是存储在一个节点上，而是按规则分布存储在一簇节点上，协同提供服务

- 写穿算法：

当用户在修改高速缓存项（文件或块）时，新的值保存在高速缓存中，并立即写回到服务器，当用户读取速缓存项（文件或块）时，需要先和服务端进行文件的 version 号比对，如果一致，直接从高速缓存项 cache 中读取文件，如果不一致则从服务器中读取文件并将文件缓存在本地高速 cache 中。

3 实验方法

本实验通过使用保证缓存一致性的写穿算法模拟了客户向分布式文件系统读写文件的过程。实验分别模拟实了高速缓存 sim_cache 类、服务器 sim_server 类、文件 sim_file 类。test_main.py 文件中代码模拟了用户读写分布式文件的过程。

3.1 读取文件

当用户使用写穿算法读取文件时，首先要判断文件是否在高速缓存中，若在高速缓存中，则需要进一步和服务器核对文件的版本号，由于服务器记录了所有文件的最新版本号，因此可以确定当前高速缓存中的文件是否最新而还未失效，若版本一致则直接从高速缓存中读取文件。否则，当高速缓存中没有文件或版本不一致时，需要通过服务器读取文件，并将文件写入本地高速缓存。读取文件的交互过程如图 4 所示。

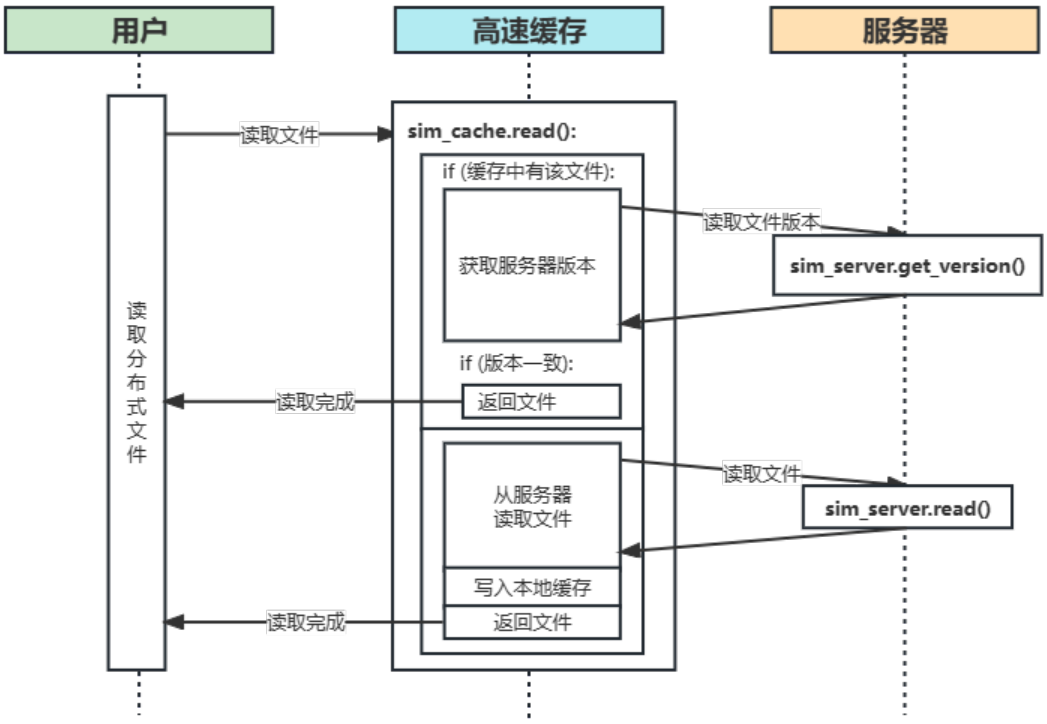

图 4 写穿算法读取文件交互过程

具体代码如下：

```
    def read(self,target_file):
        #read file from file system

        ####### Begin #######

        #检查缓存中是否已缓存文件
        cache_file = self._read_cache(target_file)
        if cache_file is not None:
            _, data, version = cache_file
            #向 server 确认 version 是否一致
            if self._target_server.get_version(target_file) ==
version:
            #选择从 cache/server 中读取文件
                return data
```

```
        #缓存从 server 中读取的文件
        _, data, version = self._target_server.read(target_file)
        self._write_cache(target_file, data, version)
        return data

        ######## End ########

        # there is no file in the system
        return
```

3.2 写入文件

当用户使用写穿算法写入文件时，首先生成一个新的版本号，然后将文件写入本地高速缓存并更新版本号，同时将文件写入服务器端并更新版本号，保证远端和本地缓存一致。写入文件的交互过程如图 5 所示。

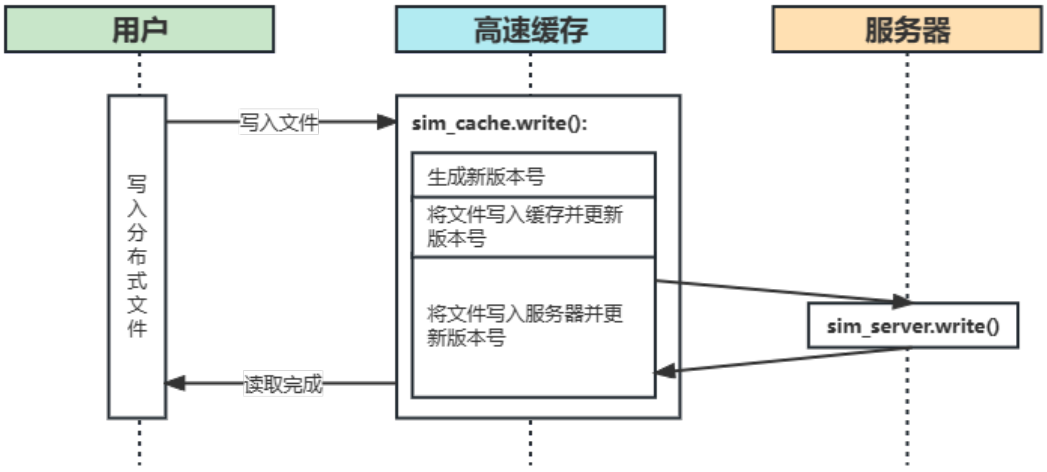

图 5 写穿算法写入文件交互过程

具体代码如下：

```
    def write(self,target_file,data):

        ####### Begin #######

        #生成新 version
        version = self._get_new_version()

        #向 cache 中写入数据并更新 version
        self._write_cache(target_file, data, version)

        #向 server 中写入数据并更新 version
        self._target_server.write(target_file, data, version)
```

```
        ######## End ########

        return
```

4 实验结果

实验在实训平台上进行，测试的输入规则和处理代码平台以及测试样例已经给出，平台会对代码进行测试。

- 测试输入解释：

第一行两个数字分别代表读写的操作次数 m 和用户数量 n。接下来的 m 行中每一行分为读和写两种操作类型，当操作类型是“write”时，后面三个参数分别是用户编号、文件编号、写入的文件数据（int 类型）；当操作类型是“read”时，后面两个参数分别是用户编号、文件编号。在程序实际运行中，不需要对输入进行处理，测试程序会处理输入并按照输入调用 cache 类的 read 和 write 函数。

- 测试输出解释：

每一行的数字是每个“read”操作后返回的文件数据（int 类型）

- 测试样例：

测试输入：

```
5 2
write 0 0 10
read 0 0
read 1 0
write 1 0 5
read 0 0
```

预期输出：

```
10
10
5
```

实验测试主要以键值对作为输入，在模拟过程中，由于读写操作是顺序的无需考虑并发等造成不一致的情况，因此高速缓存和服务器中的文件记录会保持一致，输出的即为当前文件键名对应的值的内容。以上述测试用例为例，在用户 0 写文件 0 为 10 后，用户 0 和用户 1 分别读取文件 0，都会得到 10 的结果。此后用户 1 重写文件 0 的值为 5，接着用户 0 再读取文件 0 便得到修改后 5 的结果。

如图 6 所示，代码通过了全部的测试用例。

图 6 写穿算法实训平台测试结果

总结与收获

通过本次实验，我对分布式系统中的两个关键部分：分布式共识算法以及分布式缓存一致算法有了初步的了解。Paxos 作为最为经典的分布式公式算法，其通过发起提议然后利用多数原则一步步确定提议达成共识，同时通过多个逻辑判断避免了不一致行为；而写穿算法通过添加本地高速缓存这一层，使得读取文件先经过高速缓存，减轻了服务器端网络压力并提高了读取速度。

通过实验，我也感受到分布式系统拥有多个结点，多节点达成一致统一操作是个关键问题，如何保持结点一致性减轻网络通信开销也是重要问题，当然通过课上学习我也了解到分布式系统还有容错等多方面的问题，也随之产生了许多算法、系统用于解决和处理相应的问题。通过本次实验的实践，也加深了我对课堂上相关概念知识的理解，也让我感受到了分布式系统其中的魅力与挑战，扩展了我的视野，而随着当今计算机所面对和解决的问题的复杂化与庞大化，分布式系统是解决大型问题的很重要的一方面，作为一名计算机专业的学生，我认为分布式系统是一个非常重要的技术领域，通过课堂学习和实验实践也激励了我对分布式领域的进一步学习和探索。

总之，通过《高级分布式系统》这门课程老师的专业讲解以及实验的亲身实践，我收获颇丰。