分布式数据库系统设计
项目技术报告

组长：周孟莹（19110240001）

组员：方睿钰 （19210240002）、章苏尧 （19212010032）

本报告撰写人：章苏尧

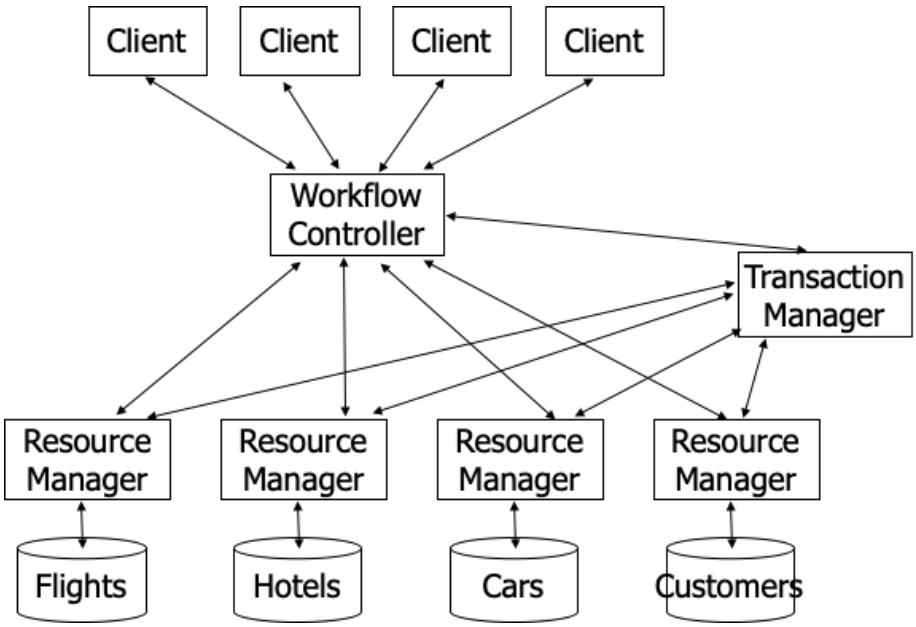
分工：

成员	分工	工作量
周孟莹 19110240001	负责协调整体项目、实现细化 Client 类、设计测试用例、撰写测试用例说明	33%
方睿钰 19210240002	实现 WorkflowController 类、编写 ReadMe 文件、撰写测试用例说明	33%
章苏尧 19212010032	实现 ResourceManager 类、调试测试用例、撰写 server 启动脚本和测试脚本	33%

一、项目设计与整体架构

本项目实现的是一个简单的分布式旅游预订系统，其中主要的数据主体为 Customer、Flight、Car、Room。Customer 可在系统上对其他三个主体进行预订和取消预订操作，另每个主体都设有增删改查的操作。

作为分布式数据系统，从设想上来看系统的各个组件以及底层数据存储都可在不同的机器上存在，因此本项目组件间的接口调用均采用 JavaRMI 模拟远程调用。本项目组件主要分为 WorkflowController（WC），TransactionManager（TM），ResourceManager（RM）三大部分，其中 WC 负责最外层的用户调用，通过与 TM 和 RM 的交互实现业务逻辑，使得 TM 和 RM 对用户透明；TM 负责管理事务，保证事务的 ACID 特性，负责事务的流程、处理错误和异常、事务回滚等；RM 则是最底层接触数据的部分，每个数据主体都由对应的资源管理器进行访问操作，实现增删改查，确保对数据持久性。



二、文件目录结构

- src
  - lockmgr: 已提供的实现好的锁管理器
  - test: Java 编写实现的测试用例
    - data: 运行生成的数据文件夹
    - TestFileObject.java: [TODO]
    - TestManager.java: 测试主程序
    - Makefile
    - 其它 Java 程序: 测试用例
  - transaction: 主要实现部分
    - data: 运行生成的数据文件夹
    - exceptions: 自定义的可能异常
    - models: 数据实体定义，包括接口 ResourceItem 与类 Car, Customer, Flight, Hotel, Reservation, ReservationKey
    - Client.java: 脚本测试时调用的客户端
    - MyClient.java: 默认简单客户端
    - ResourceManager.java: RM 接口
    - ResourceManagerImpl.java: RM 接口实现
    - RMTable.java: 数据表
    - TransactionManager.java: TM 接口（主要修改文件）
    - TransactionManagerImpl.java: TM 接口实现（主要修改文件）
    - Utils.java: 工具类，用于简化代码
    - WorkflowController.java: WC 接口
    - WorkflowControllerImpl.java: WC 接口实现（主要修改文件）
    - Makefile
- doc
  - 测试用例说明
  - 项目分工说明
  - 项目技术报告-周孟莹
  - 项目技术报告-方睿钰
  - 项目技术报告-章苏尧
- README.md
- run_server.sh
- run_test.sh
- stop_server.sh

三、需求分析

我负责的是事务管理器这个模块，其主要操作对象是事务，目的是维持事务的 ACID 特性，难点是系统执行过程中，可能遇到各种出错，比如 RM 崩溃，WC 崩溃，TM 自己崩溃等，要设计通过合理的冗余性避免崩溃带来的的影响，并能够恢复已经提交的事务。

四、事务管理器的设计与实现

在分布式事务中，一般使用发号器提供全局唯一事务号。由于本项目事务管理器只有一个，简化后可以简单的使用内存变量提供全局唯一的事务号，利用 Java 的 synchronized 关键字，在每次调用写事务号的时候给此变量加锁，以此保持其唯一性。

事务管理器主要数据结构如下：

```
// in single tm, using xidNum with file record can ensure uniquness
private Integer xidNum;

// tm's properties, used to simulate crash
private String dieTime;

// all active transactions and their status
private HashMap<Integer, String> xidStatus;

// active transaction and their related rms
private HashMap<Integer, HashSet<ResourceManager>> xidRMs;
```

其中，xidNum 是之前提到的全局唯一事务号，dieTim 略过。重点关注 xidStatus 和 xidRMs 两个哈希表。他们的主键都是事务号，其值分别是事务对应的状态和参与事务的 RM 的集合。

对于一个事务，本项目设置四个状态：inited、prepared 和 committed 和 aborted。除了 commited 阶段，其他发生了错误都要回滚。

事务管理器在事务管理方面主要实现了以下五个接口：

- start
  - 由 wc 调用，主要目的是生成全局唯一事务号，并配置磁盘写持久化事务号，以供崩溃恢复的时候读取
- ping
  - 此接口实现可以只是一个空函数，主要目的是抛出 RemoteException，目的是用来检测与 TM 的连接是否失效。在 Java RMI 模拟远程调用的情况下，如果与 TM 的连接失效，调用此函数会抛出异常。
- abort
  - 用来终止事务。会遍历内存中存储的对应事务号的所有的 RM 引用，调用他们的 abort 方法，在两个哈希表中删除对应的条目，之后持久化存储 xidStatus 和 xidRMs。这个过程中可能遇到要调用的 RM 引用已经失效的问题。但是没有关系，这种情况不用做特别处理。因为一旦从两个哈希表中删除以后，此事务号无法再从 commit 和 enlist 访问到
- commit
  - commit 中使用集中式的两阶段提交处理。两阶段提交保证了分布式事务的原子性。具体的过程就不赘述了，简单而言，遍历查询-切换到 prepare/abort-遍历 rm 执行 commit。在这个过程中，可以认为步骤 3 阶段的数据是必须要持久化的，无论遇

到什么崩溃，都要确保提交过的事务的一致性。commit 方法执行到最后也会操作两个哈希表，主要目的是移除已经 committed 的事务。这个过程中可能会出现并不是所有的 rm 在 committed 阶段都会正确执行自身的 commit 方法。为此，会删去那些确认 commit 过的方法，保留没有执行提交的 rm 的集合，之后会在 enlist 方法中处理这种情况。在 enlist 中处理是因为一个已经崩溃了 RM，在它重启想要操作的时候必须要通过 enlist 方法让 TM 获得其引用

- enlist
  - enlist 方法是很重要、很有技巧的一个方法。也是我做这个项目最大的收获。如我上面所说，enlist 方法的目的主要是让 TM 获得 RM 的引用和一些出错情况的处理。异常情况包括：
  - 已经转到 committed 状态的事务，但是其 RM 并没有正确执行 commit 方法就崩溃，此 RM 恢复过后，会想知道此事务的状态。在我们的模型中，RM 会直接通过 enlist 方法询问 TM
  - 所有的非 committed 状态的事务在 TM 重启过后都会标记为 aborted。此时这些事务中的 RM 访问会触发 abort。
  - 上面两种异常情况处理过后，开始 enlist 的正常功能处理，即让 TM 获得并保存 RM 的引用。此时一是要确保这个过程 RM 不崩溃，二是要确保已经 enlist 的 RM 不会再次 enlist

```
synchronized (xidRMs) {
    // if RMDieAfterEnlist, should abort this transaction;
    // random rm may die, so should check every rm
    boolean reEnlist = false, rmDieAfterEnlist = false;
    for (ResourceManager checkRM : xidRMs.get(xid)) {
       try {
          if (checkRM.getID().equals(rm.getID())) {
             reEnlist = true;
          }
       } catch (Exception e) {
          rmDieAfterEnlist = true;
          break;
       }
    }

    if (rmDieAfterEnlist) {
       this.abort(xid);
       return;
    }

    if (!reEnlist) {
       xidRMs.get(xid).add(rm);
       Utils.storeObject(TM_XID_RMS_PATH, xidRMs);
    }
 }
```

- recover
    - 这是 TM 的私有方法，主要用来从文件中读取之前持久化的数据并恢复。

至此，TM 设计完毕。

五、其他心得

Q：Lock 和 synchronized 的区别在哪儿？什么时候使用提供的锁，

什么时候使用 synchronized？

A：Lock 提供了更细粒度的管理，方便减少花销，提升系统性能。

不一定所有的时候都是互斥访问的，比如读读访问，这种时候用 synchronized 就会导致即使是读读访问，也会保证互斥性，使用自定义的锁，确保类似的非互斥访问可以执行，互斥访问不能执行。