TONGJI UNIVERSITY

毕业实训

课题名称 多数据中心场景的大规模分布式任务及数据联合调度模拟器构建

摘 要

云计算是一种按需提供计算资源的成熟技术，目前面临着诸多挑战。主要挑战包括共享资源的管理、能源消耗、负载平衡、资源供应和分配等。云模拟器可以使得研究者不需要使用真实的云计算环境就可以实验新的模型和算法，为研究提供便利。本次开发的多数据中心场景的大规模分布式任务及数据联合调度模拟器是一个开源模拟器，它基于 cloudsim plus 开发，具有强可拓展性和实用性。它能够在多个服务器上同时运行，每个服务器当做一个数据中心，各个服务器之间相互协作共同完成同一个模拟任务。同时它通过 yml 文件来描述数据中心的各项配置还有客户的各种需求，不需要修改代码就能够完成大多数场景的模拟。本文后面详细介绍了代码的结构，主要的类还有运行的时序图，并且通过实验来详细分析了程序运行时间的占比分析，还对代码中提供的三种基础的客户需求分配算法进行了实验验证，分别是随机分配、资源花费最低分配和循环适配，通过设计实验和对结果的分析证明了他们的正确性。通过这次的研究也为后续毕业设计的算法研究打下在坚实的基础。

关键词：

云计算，cloudsim plus，分布式模拟

ABSTRACT

Cloud computing is a mature technology that provides computing resources on demand, but it is currently facing many challenges. The main challenges include management of shared resources, energy consumption, load balancing, resource provisioning and allocation, etc. The cloud simulator allows researchers to experiment with new models and algorithms without using a real cloud computing environment, which facilitates research. The large-scale distributed task and data joint scheduling simulator for multi-data center scenarios developed this time is an open source simulator, which is developed based on cloudsim plus and has strong scalability and practicability.It can run on multiple servers at the same time, each server is regarded as a data center, and each server cooperates with each other to complete the same simulation task. At the same time, it uses yml files to describe the configuration of the data center and various needs of customers, and can complete the simulation of most scenarios without modifying the code. Later in this paper, the structure of the code is introduced in detail, the main classes and the timing diagram of the operation, and the proportion analysis of the program running time is analyzed in detail through experiments, and the three basic customer demand allocation algorithms provided in the code are also analyzed. Experimental verification, random allocation, minimum resource cost allocation and cyclic adaptation, proved their correctness by designing experiments and analyzing the results. This research has also laid a solid foundation for the algorithm research of the subsequent graduation project.

Key words：

cloud computing, cloudsim plus, distributed simulation

目 录

1 引言

1.1 课题背景

模拟器是一种以可接受的误差来重现特定系统的行为的软件，目前已经广泛用于各个科学领域。在大多数情况下，科学家如果想要复现某种真实事件或者实际使用某些机器、部署某种系统都会面临费用昂贵、操作不易、时间久、结果难以复现等困难。所以尽管模拟器难以重现复杂的真实世界的系统，但他使用更加简便快捷，系统行为有迹可循，结果方便复现，是辅助科学研究的一大利器。

云计算可以按需提供服务，降低企业成本，具有高可拓展性、高可靠性，因此今年来不仅收到工业界的追捧也受到科研界的重视，如何降低能源成本、如何充分利用资源、如何高效完成服务一直都是研究的重点。但是如果想要在真正的云计算系统上实践科学研究是非常难以实现且价格高昂的，所以现在对云计算的科学研究一般都借助云计算模拟器来进行模拟研究。

目前已经有一批云计算模拟器对推出，他们不仅包括传统的云计算模拟还包括针对新兴的雾计算和边缘计算的模拟。现有的模拟器的主要问题是可拓展性弱，他们都是在单个服务器上模拟，而单个系统的性能很容易就能达到上限，因此可模拟的服务器和任务的数量有限，难以完成大规模的模拟。同时本模拟器是为了后续的大规模分布式集群资源调度算法研究做准备。不仅要求大规模还要求分布式，因此参照这些要求，开发设计了这个大规模分布式模拟器。它的主要特点如下：

1. 按照一个数据中心为粒度，模拟器可以在一个服务器上模拟一个数据中心，多个服务器之间可以协调合作，共同完成多个数据中心的模拟。
2. 每个数据中心有自己的分布式决策系统，根据自己获取的信息来决策发给自己的请求是本数据中心完成还是交给其他哪个数据中心。
3. 采用 yml 文件的方式来描述数据中心的资源价格、主机配置，还有用户的需求等等。

可以通过如下链接来获取全部源代码：https://github.com/slipegg/DistributedCloudsim

1.2 研究现状

云计算模拟器在过去十年间出现了数十种，我这里主要将其分为云计算模拟器和边缘计算模拟器两类。云计算模拟器就是我们本次要做的模拟器，而边缘计算模拟器一般都是由云模拟器改编而来，具有多层分布式决策的特点，具有一定的参考意义。

云计算模拟器

Cloudsim

Cloudsim 在 2011 年提出，它建立在 GridSim 之上，是第一个旨在模仿一般云行为的模拟器。它提供了一个通用的、可定制的模拟工具来模拟大多数的云功能，包括服务代理、任务调度、VM 分配等等，它也经常被后面的模拟器的开发者当做一个模拟器的核心组件来使用。

Cloudsim Plus

CloudSim Plus 项目在 2017 年作为 CloudSim 的一个独立分支发起，它对 Cloudsim 进行

了重构，改进了类层次和代码，使得代码更加易于使用和理解，还增添了许多实用的新功能，并且提供了全面的详细的文档，包括使用示例、测试示例、api 文档等等。本模拟器的开发也是依托于它来完成的。

Cloudsimscale

CloudSimScale 扩展了 CloudSim 以在分布式环境中运行数据中心的对象。该模拟器的作者使用 IEEE 高级架构 (HLA) 和运行时基础架构 (RTI) 使原始 CloudSim 组件能够在分布式执行期间进行通信，主要的扩展类是 DatacenterBroker、Datacenter 和 CIS。这一个模拟器是和我要做的模拟器类似的，但是它最大的问题是没有开源的代码，导致无法使用这一个模拟器，同时它借助 HLA 来实现组件间的通信，其实现比较复杂且效果有局限，所以本次就自己开发了一个分布式的模拟器。

边缘计算模拟器

iFogsim2

iFogsim2 是同一个作者开发的 iFogsim 的第二版，发表于 2022 年，而 iFogsim 一直是引用数最多的模拟器，这一次推出的第二版参考了过往边缘模拟器的许多优缺点来对 iFogsim 进行改进，它不仅可以模拟执行延迟、能源消耗、网络拥塞等情况，以评估资源管理和调度策略，还支持分布式调度、支持微服务和集群模拟。

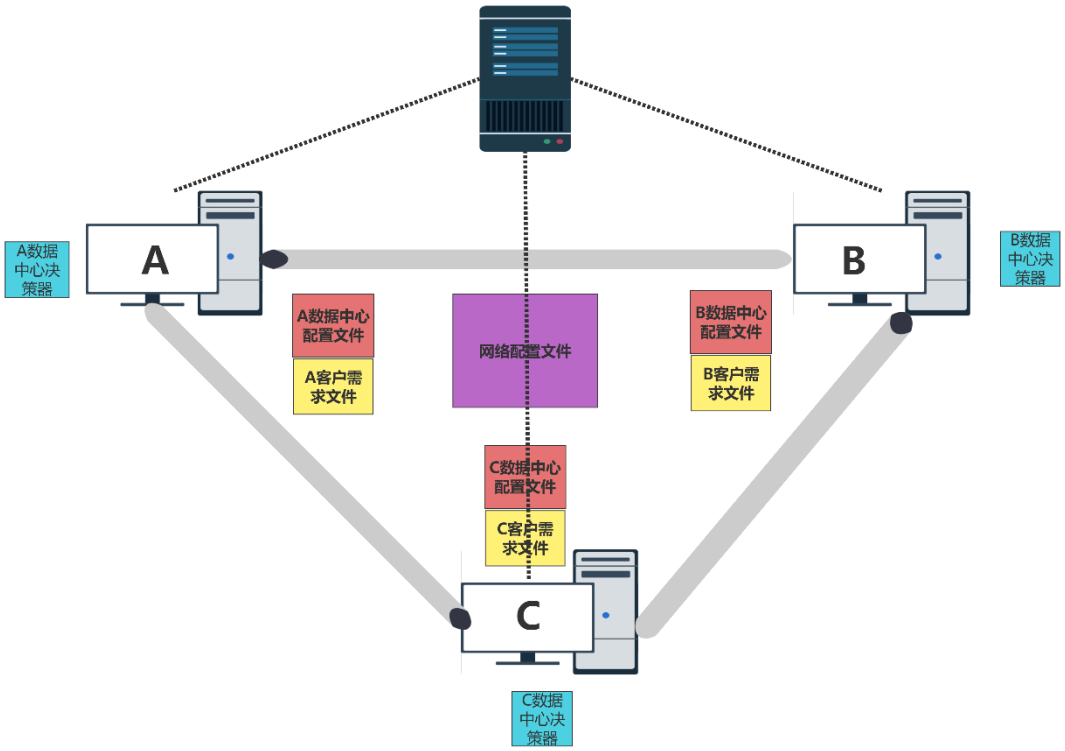
PureEdgeSim

PureEdgeSim 可以模拟边缘计算和云计算的所有层，并且使用模块化设计，支持设备和网络的异构性，可以使用现实的网络模型和移动模型，支持在线决策，提供了广泛的结果指标，还对模拟进行了实时的可视化，具有高可拓展性，用户无需修改代码就可以实现自己场景的模拟，并且通过移除最初的 cloudsim plus 转而自己实现事件队列来使得模拟速度变得更快。

2 研究内容与方法

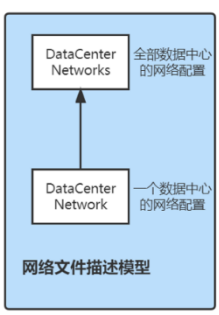
2.1 整体概述

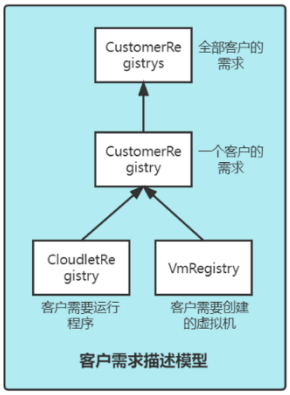
本模拟器使用 java17 进行构建，主要基于 cloudsim plus 这一个库。它的整体架构图如下：

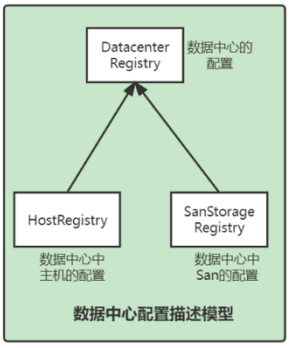


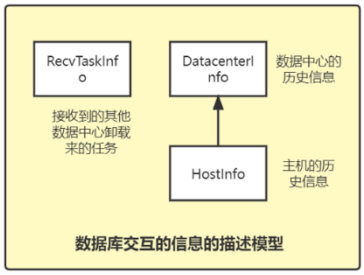
每个服务器单独运行一个模拟器来模拟一个数据中心，每个服务器有自己的数据中心配置文件和客户需求文件，然后这些文件都是每个服务器上都有一份的，同时还共同拥有一份网络配置文件，每个服务器上的数据中心的决策器都可以以及自己获得的讯息来进行分布式的决策。服务器之间的信息沟通和同步有两种方式，一种是借助 socket 连接的网络通信，这里的作用主要是用来同步各个服务器模拟的时间，还有一种是借助数据库，每个服务器共同连接到同一个数据库，将自己数据中心的状态还有要转移出去的任务放到数据库中去。

整体创建的基础数据类型的模块如下：



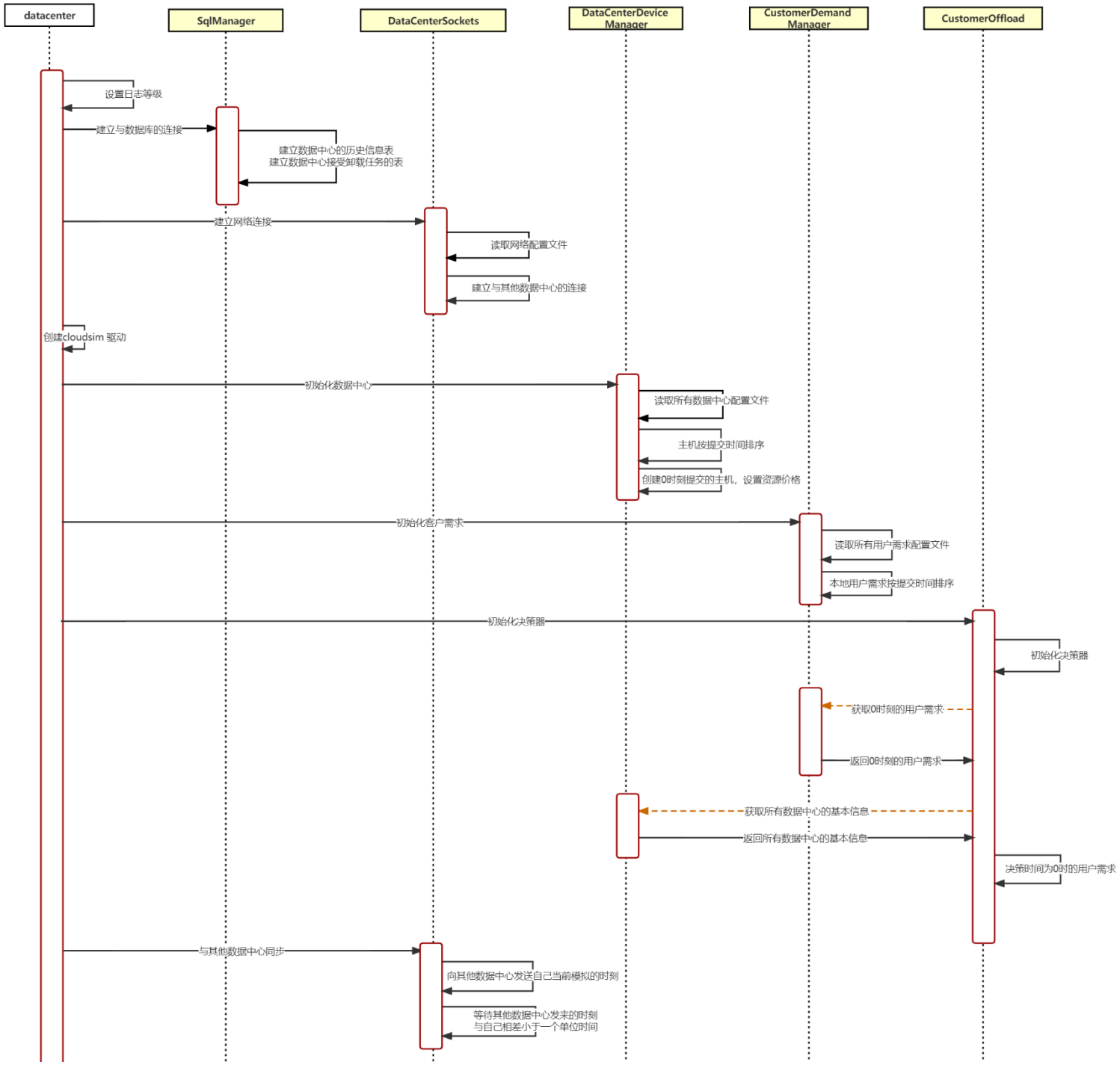


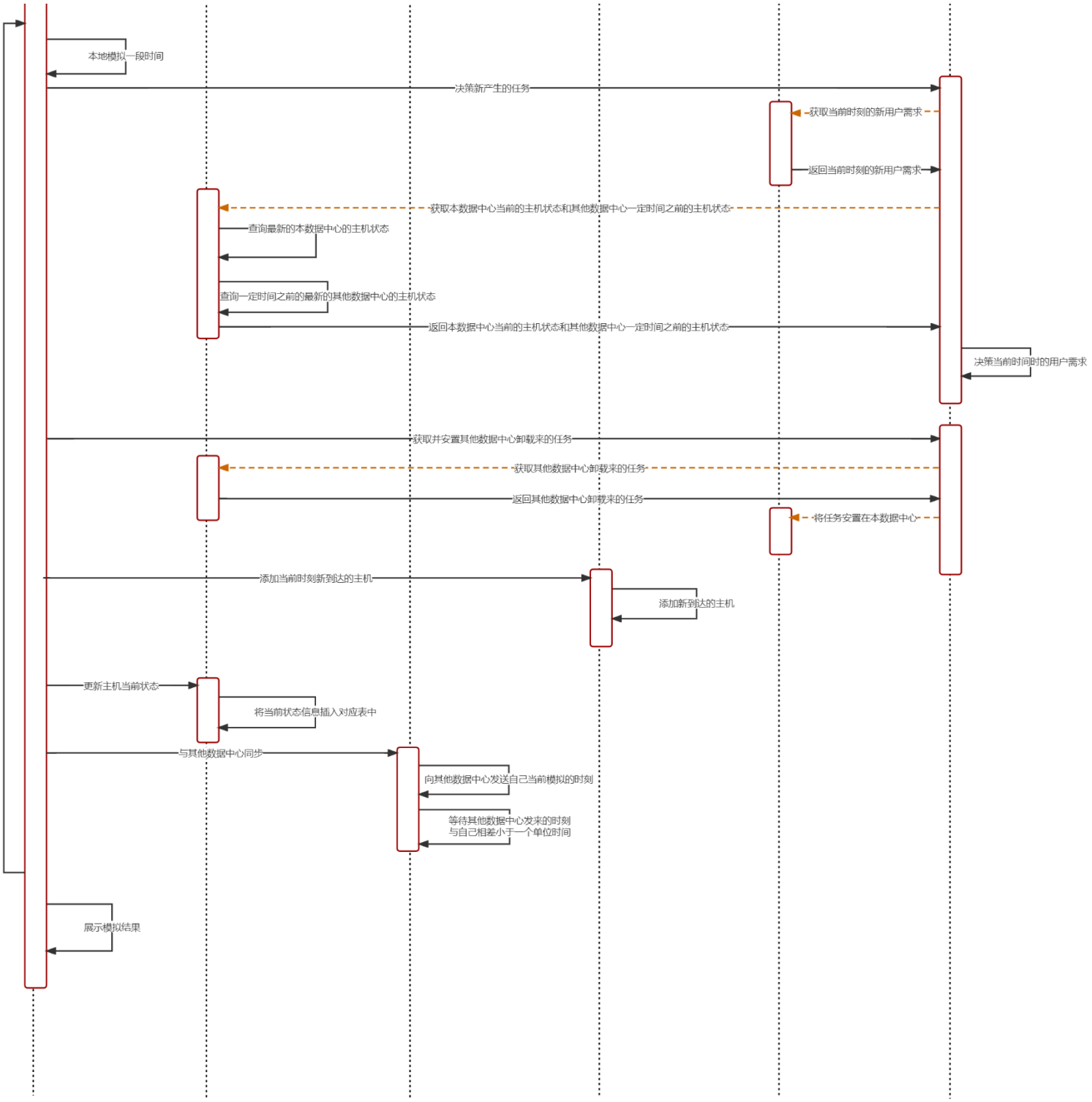




好了网络文件描述模型、客户需求描述模型还有数据中心配置描述模型，并且对于与数据库交互过程中接收到的其他数据中心发来的任务还有其他数据中心的历史信息都进行了模型规范。每一个模型基本都由多个子模型组成，以客户需求描述模型为例，一个客户需要运行的程序和客户需要创建的虚拟机共同组成了一个客户的需求，而多个不同的客户需求就组成了全部用户的需求，而这就是客户需求文件里描述的内容。其他的也是类似。

程序运行的整个流程如下所示：





整个程序运行主要由 SqlManager、DataCenterSockets、DataCenterDeviceManager、CustomerDemandManager、CustomerOffload 这几个核心运行的类组成。在程序一开始时，我们需要设置日志等级，主要有 DEBUG、INFO、WARNING、ERROR 这几种。然后我们需要建立与数据库的连接，这里使用的是连接池的做法来与数据库维持连接，建立连接后还需要建立属于本数据中心的两张表，即数据中心的历史信息表和数据中心接受卸载任务的表。然后我们需要创建并等待与其他数据中心的网络连接，这里使用的是反应迅速的异步 NIO。然后我们创建模拟核心的 cloudsim 驱动。然后初始化本数据中心，初始化依靠的

是 device.yml 文件，通过读取文件内容获得资源价格和主机数量和配置，加载 DatacenterRegistry 这个类，然后对于所有的主机按照提交时间来进行排序，将 0 时刻提交的主机全部创建起来，并设置好资源的价格。然后初始化客户需求，初始化依靠的是 customer.yml 文件，通过读取文件内容获得每一个客户需求的虚拟机还有要在虚拟机上执行的任务，并加载出 CustomerRegistrys 这个类，并对所有的客户需求按照提交时间进行排序。然后再初始化决策器，建立决策器之后去获取 0 时刻的客户需求，0 时刻没有任何历史信息，所以决策器决策依靠的是所有数据中心的基本信息来决策当前 0 时刻的用户需求，如果是在本数据中心完成该任务，就将其加入到任务队列中，不然就是卸载到其他数据中心，就需要将任务存储到对应的数据库的表中。然后在准备开始模拟之前，还需要进行与其他数据中心的同步，同步的做法是本地数据中心将自己当前模拟的时刻发送给其他数据中心，同时自己来不断查看自己接收到的其他数据中心发来的模拟时刻，当确保所有其他数据中心都没有比自己慢超过 1 个时间单位的时候，就说明同步完成了。

在开始模拟时，本地先模拟一小段时间，然后决策器再查看是否有新的客户需求到达了，如果有新的客户需求到达了就去查询数据库来获得本数据中心最新的主机状态信息，还需要去查询一定时间延迟之前的最新的其他数据中心的主机的状态信息，然后再还有就是我们需要的其他信息来决策当前新来的客户需求需要放置到哪个数据中心。然后还需要去查询数据库来获取其他数据中心卸载来的任务，并需要将这些任务放置在本数据中心中。然后就还需要看是否有新到达的主机需要添加到本数据中心中。这一切介绍之后就需要更新主机当前的状态，将当前的状态信息插入到对应的历史信息表中，然后再次进行与其他数据中心的时间同步。然后就继续模拟一小段时间，重复上述过程，直到到达模拟结束时间，最后就需要展示模拟结果。

2. 2 模块介绍

这里介绍 SqlManager、DataCenterSockets、DataCenterDeviceManager、CustomerDemandManager、CustomerOffload 这几个核心的模块。

SqlManager

SqlManager 负责数据库相关的操作，这里主要为更新和获取数据中心主机的历史信息，还有更新和获取数据中心卸载的任务。

SqlManager 在初始化时会借助 hikari 库来创建连接池，有了连接池之后在每次建立与数据库的连接的时候就不需要重新耗时耗力地建立连接而是直接从存放了一定数量的与数据库服务器连接好的连接池中取出一条连接来即可。由此就可以大大减少连接数据库的开销，从而提高性能。

数据中心主机历史信息表如下：

Field	Type	Null	Key	Default	Extra
id	int(10) unsigned	NO	PRI	NULL	auto_increment
host_id	int(11)	NO		NULL	
run_time	double	NO		NULL	
ram	int(11)	NO		NULL	
bw	int(11)	NO		NULL	
storage	int(11)	NO		NULL	
free_pes	int(11)	NO		NULL	
free_mips	double	NO		NULL	

主机的历史信息表的表名规定为：数据中心名_history。每次数据中心初始化 SqlManager 的时候就建立其对应的这种表。这张表的内容包括信息的 id，主机的 id，当前历史信息对应的时间，主机当前剩余的内存空间，主机当前剩余的宽带，主机当前剩余的存储空间，主机当前空余的核，主机当前剩余的计算能力。每次数据中心更新历史信息的时候，都需要遍历本数据中心的所有主机，获得每个主机的这些状态，然后将其发送出去并更新数据库。而在获取这些历史信息的时候，对于获取本数据中心的主机历史信息的，只需要先查询得到最新的时间，然后依据这个时间将所有主机的历史信息取出来即可，但是对于查询其他数据中心的历史信息，这里就不会直接查询最新的时间，而是在一定的延迟时间前的最新的时间的历史信息，然后再依据这个时间取出所有主机的历史信息。

数据中心接收到的其他数据中心卸载过来的客户需求的表如下：

Field	Type	Null	Key	Default	Extra
id	int(10) unsigned	NO	PRI	NULL	auto_increment
datacenter_name	varchar(255)	NO		NULL	
task_index	int(11)	NO		NULL	
amount	int(11)	NO		NULL	
send_time	double	NO		NULL	
is_get	tinyint(1)	YES		0	

这个表的表名规定为：task2 数据中心名。每次数据中心初始化 SqlManager 的时候也会建立其对应的这种表。这张表的内容包括：信息的 id，客户需求来源的数据中心名，客户需求的序号，这同一个需求的数量，客户需求送达的时间，是否以及被取走安置了。这里的序号指的是按提交时间排序过后的客户需求的序号，由于每个数据中心都有其他数据中心的客户需求文件，并且排序都是依靠提交时间，所以简单地依靠序号就可以得到是哪个客户需求了。每次模拟一段时间后，SqlManager 就需要来查看一下当前这张表是否有 is_get=0 的新来的客户需求，然后就需要将其取走，并将 is_get 置为 1。

DataCenterSockets

DataCenterSockets 负责网络连接，主要是为了快速同步各个数据中心的模拟时钟。在初始化时，DataCenterSockets 会建立一个新线程，这个线程运行 NIO，即非阻塞连接，它有一个多路复用接口 selector，阻塞同时监听来自多个客户端的 IO 请求，一旦有收到 IO 请求就调用对应函数处理，NIO 擅长 1 个线程管理多条连接，可以有效节约系统资源。它主要负责快速建立连接和快速接受信息，一旦有连接请求建立 selector 就会立即搭建 socket 连接，后续对应的数据中心就可以依靠这一个 socket 连接来发送当前模拟的时刻。而在对应的信息发送过来之后，selector 就会取出该信息，然后将信息更新到对应的变量中去。除了这个新线程，DataCenterSockets 还会不断尝试与其他数据中心建立连接，直到建立成功，后续就会依据这个建立的 socket 来发送本数据中心当前模拟的时刻。

发送信息的格式如下：

数据中心名 time 当前模拟的时刻\n

例如一个名为 c1 的数据中心，当前模拟时刻为 10.2，那么发送的信息就是：c1 time 10.2\n。这样接受到的就能够知道 c1 数据中心模拟到了 10.2 秒的时刻。但是注意在接收信息时有可能出现同一个数据中心的多条信息紧接在了一起，所以就需要依据\n 的回车符来将信息分割，并取最后的信息当做该数据中心最新的模拟时刻。

需要同步的原因是如果不同步的话，有可能 a 数据中心要在 10 秒的时候将任务卸载到 b 数据中心，但是此时 b 数据中心依据模拟到了 30 秒甚至已经模拟结束，这样子将任务卸载过去的意义就不对了，所以需要控制每个数据中心的模拟时间，这里规定数据中心之间模拟的时间差距最大不能超过 1 秒。每次在同步时，DataCenterSockets 先将自己的当前的模拟时刻发送给其他数据中心，然后去查看自己 selector 接收到的其他数据中心的模拟的时刻，直到接收到的模拟时刻没有比自己慢超过 1 秒的存在就算同步完成。

DataCenterDeviceManager

DataCenterDeviceManager 主要是依靠读取 yml 文件来加载配置 DatacenterRegistry 类，然后根据这个 DatacenterRegistry 类来设置模拟器中资源的价格，还有模拟器中数据中心的主机。除了读取本数据中心文件之外，还需要读取其他数据中心的文件，并转化为 DatacenterRegistry 类将其存储起来。同时需要注意在加载了 Host 主机之后需要按照提交时间来进行排序，这样方便后面来将最新提交的主机加入到数据中心中。

DatacenterRegistry 类的定义如下：

```
public final class DatacenterRegistry implements Serializable {
    private long id;
    private String name;
    private Integer amount;
    private String architecture;
    private String os;
    private String vmm;
    private double timeZone;
    private String vmAllocationPolicy;
    private boolean vmMigration;
    private List<HostRegistry> hosts;
    private double costPerSec;
    private double costPerMem;
    private double costPerStorage;
    private double costPerBw;
    private List<SanStorageRegistry> sans;
    private double upperUtilizationThreshold;
    private double lowerUtilizationThreshold;
    private double schedulingInterval;
```

主要需要定义的有：数据中心的 id，数据中心的名字，数据中心的数量，架构类别，操作系统，虚拟机类别，时区，虚拟机分配策略，虚拟机迁移策略，主机属性，每秒花费的价格，每单位内存使用的价格，每单位存储空间使用的价格，每单位宽带使用的价格，

san 网络，资源最高使用率，资源最低使用率，调度时间价格。

其中主机属性的定义如下：

```
public final class HostRegistry implements Serializable, Comparable<HostRegistry> {
    private int id;
    private double submit_time;
    private int pes;
    private double mips;
    private double maxPower;
    private double staticPowerPercent;
    private int ram;
    private long bw;
    private String ramProvisioner;
    private String bwProvisioner;
    private String peProvisioner;
    private String vmScheduler;
    private String powerModel;
    private int amount;
    private long storage;
```

Host 主要需要定义的元素有主机 id，主机提交时间，pe 核数量，pe 核的运行能力，最大使用电量，静态使用电量率，内存，宽带，内存使用策略，宽带使用策略，pe 核使用策略，vm 虚拟机共享配置，数量，存储。

数据中心的模拟主要依靠的是 Cloudsim Plus 的模拟，依据 DatacenterRegistry 的内容来定义 Cloudsim Plus 中对应 DatacenterSimple、Host 类的属性。

CustomerDemandManager

CustomerDemandManager 主要也是依靠 yml 文件来加载配置 CustomerRegistrys 类，注意加载好的 CustomerRegistrys 类中的 CustomerRegistry 也需要按照提交时间来进行排序。除此之外它提供的方法有得到当前时刻提交的客户需求，依据客户需求在数据中心主机上创建对应的虚拟机还有任务，根据其他数据中心卸载来的客户需求在本数据中心创建客户需要的虚拟机还有任务。

CustomerRegistrys 类定义如下：

```
public class CustomerRegistrys implements Serializable {
    private List<CustomerRegistry> customers;
```

CustomerRegistry 类定义如下：

```
public final class CustomerRegistry implements Serializable, Comparable<CustomerRegis
    private long id;
    private String name;
    private Integer amount;
    private double submit_time;
    private List<VmRegistry> vms;
    private List<CloudletRegistry> cloudlets;
    private String preference;
```

CustomerRegistry 主要需要定义的属性有客户 id，客户姓名，数量，提交时间，客户需要的虚拟机，客户需要在虚拟机上执行的任务，客户的偏好。

VmRegistry 的定义如下：

```
public final class VmRegistry implements Serializable{
    private long id;
    private long size;
    private int pes;
    private double mips;
    private int ram;
    private long bw;
    private int priority;
    private String vmm;
    private String cloudletScheduler;
    private int amount;
```

主要需要定义的属性有虚拟机 id，虚拟机大小，虚拟机 pe 核的数量，虚拟机 pe 核的计算能力，虚拟机需要的内存，虚拟机需要的宽带，虚拟机的优先级，虚拟机的类别，虚拟机的任务放置策略，该类别虚拟机的数量。

CloudletRegistry 类的定义如下：

```
public final class CloudletRegistry implements Serializable {
    private long id;
    private double timeZone;
    private int amount;
    private long length;
    private long fileSize;
    private long outputSize;
    private int pes;
    private String utilizationModelCpu;
    private String utilizationModelRam;
    private String utilizationModelBw;
    private double submissionDelay;
```

主要定义的属性有：任务 id，任务所在的时区，任务的数量，任务的长度，任务所需要的文件大小，任务输出的文件的大小，任务需要的 pe 核的数量，任务的 CPU 使用策略，任务的内存使用策略，任务的宽带使用策略，任务的提交延迟。

客户的模拟也主要依靠的是 Cloudsim Plus 的模拟，依据 CustomerRegistrys 类的属性来定义 Cloudsim Plus 中 Broker、Vm、Cloudlet 这些对应类的属性。

CustomerOffload

CustomerOffload 的主要作用是依据一定的规则来分配本数据中心的客户需求。目前已经完了随机分配，资源花费最低分配、循环适配这 3 种分配方式。

随机分配代码如下：

```
void randomOffload(CloudSim simulation, double now_time) {
    int runned_customer_index = customer_manager.getRunned_customer_index();
    CustomerRegistrys customerRegistrys = customer_manager.getNowCustomerRegistrys(now_time);

    Random r = new Random();
    for (int i = 0; i < customerRegistrys.getCustomers().size(); i++, runned_customer_index++) {
        CustomerRegistry customer = customerRegistrys.getCustomers().get(i);
        for (int j = 0; j < customer.getAmount(); j++) {
            index = r.nextInt(1 + other_datacenter_name.size());
            System.out.println("random index:" + Integer.toString(index));
            if (index == 0) {
                System.out.println(
                        customer.getName() + "第" + Integer.toString(j) + "个，卸载在本地数据中心" + datacenter_
                customer_manager.createVmsAndCloudletsLocallyByCustomerRegistry(customer, simulation, j);
            } else {
                System.out.println(
                        customer.getName() + "第" + Integer.toString(j) + "个，卸载到其他数据中心"
                                + other_datacenter_name.get(index - 1) + "\n");
                sqlManager.insertTaskIntoTable(other_datacenter_name.get(index - 1), runned_customer_index, ar
                        now_time);
            }
        }
    }
```

首先是需要获得当前时刻提交的所有客户需求，然后遍历这些客户需求，每次遍历时产生一个 0-所有数据中心个数的随机数，如果随机数是 0 就分配给本数据中心，调用 CustomerManager 相关的 createVmsAndCloudletsLocallyByCustomerRegistry 方法来在本地数据中心完成这个客户需求，如果随机数是其他就说明需要卸载到其他数据中心，就调用 SqlManaager 相关的 insertTaskIntoTable 方法来将这个客户需求放置到对应的数据中心接收表中。

资源花费最低分配如下：

```
void cheapesOffload(CloudSim simulation, double now_time) {
    int runned_customer_index = customer_manager.getRunned_customer_index();
    CustomerRegistrys customerRegistrys = customer_manager.getNowCustomerRegistrys(now_time);

    for (int i = 0; i < customerRegistrys.getCustomers().size(); i++, runned_customer_index++) {
        CustomerRegistry customer = customerRegistrys.getCustomers().get(i);
        for (int j = 0; j < customer.getAmount(); j++) {
            double min_price = 0;
            min_price = calculate_price(datacenter_manager.datacenterRegistry, customer);
            String cheapest_datacenter_name = datacenter_manager.getDatacenter_name();
            for (String datacenter_name : other_datacenter_name) {
                DatacenterRegistry tmp = datacenter_manager.other_datacenterRegistries.get(datacenter_name);
                double res = calculate_price(tmp, customer);
                if (res < min_price) {
                    min_price = res;
                    cheapest_datacenter_name = datacenter_name;
                }
            }
            if (cheapest_datacenter_name.equals(datacenter_manager.getDatacenter_name())) {
                System.out.println(
                        customer.getName() + "卸载在本地数据中心" + datacenter_name + "\n");
                customer_manager.createVmsAndCloudletsLocallyByCustomerRegistry(customer, simulation,
                        j);
            } else {
                System.out.println(
                        customer.getName() + "卸载到其他数据中心"
                                + cheapest_datacenter_name + "\n");
                sqlManager.insertTaskIntoTable(cheapest_datacenter_name, runned_customer_index,
                        customer.getAmount(),
                        now_time);
            }
```

首先依旧是需要获得当前时刻提交的所有客户需求，然后遍历这些客户需求，然后计算在每个书中中心下分配这些资源所需要的价格，计算的代码如下：

```
private double calculate_price(DatacenterRegistry datacenterRegistry, CustomerRegistry customer) {
    double costPerMem = datacenterRegistry.getCostPerMem();
    double costPerStorage = datacenterRegistry.getCostPerStorage();
    double costPerBw = datacenterRegistry.getCostPerBw();

    double price = 0.0;
    for (VmRegistry vm : customer.getVms()) {
        price += vm.getAmount()
                * (vm.getRam() * costPerMem + vm.getSize() * costPerStorage + vm.getBw() * costPerBw);
    }
    return price;
}
```

这里主要考虑的是内存、存储空间还有宽带的费用：

*价格 = 数量 * （内存大小 * 内存单价 + 虚拟机大小 * 存储单价 + 宽带大小 * 宽带单价）*

在计算好了价格之后比较得到价格最低的数据中心，如果这个价格最低的数据中心是本数据中心就调用 CustomerManager 相关的 createVmsAndCloudletsLocallyByCustomerRegistry 方法来在本地数据中心完成这个客户需求，如果是其他数据中心，就调用 SqlManaager 相关的 insertTaskIntoTable 方法来将这个客户需求放置到对应的数据中心接收表中。

循环适配代码如下：

```
void suitOffload(CloudSim simulation, double now_time) {
    int runned_customer_index = customer_manager.getRunned_customer_index();
    CustomerRegistrys customerRegistrys = customer_manager.getNowCustomerRegistrys(now_time);
    for (int i = 0; i < customerRegistrys.getCustomers().size(); i++, runned_customer_index++) {
        CustomerRegistry customer = customerRegistrys.getCustomers().get(i);
        for (int j = 0; j < customer.getAmount(); j++) {
            if (isSuit(datacenter_manager.getDatacenter_name(), customer, now_time, isLocal: true)) {
                System.out.println(
                        customer.getName() + "卸载在本地数据中心" + datacenter_name + "\n");
                customer_manager.createVmsAndCloudletsLocallyByCustomerRegistry(customer, simulation,
                        j);
            } else {
                for (String datacenter_name : other_datacenter_name) {
                    if (isSuit(datacenter_name, customer, now_time, isLocal: false)) {
                        System.out.println(
                                customer.getName() + "卸载到其他数据中心"
                                        + datacenter_name + "\n");
                        sqlManager.insertTaskIntoTable(datacenter_name, runned_customer_index,
                                amount: 1,
                                now_time);
                    }
                }
            }
        }
    }
```

同样的，首先依旧是需要获得当前时刻提交的所有客户需求，然后遍历这些客户需

求，看当前这个客户需求是否适合在本地数据中心放置，如果不适合就循环遍历看适合哪一个数据中心，并将其分配过去，如果都不适合就直接丢弃了。

判断是否适合的代码如下：

```
private boolean isSuit(String datacenter_name, CustomerRegistry customer, double now_time, boolean isLocal) {
    DatacenterInfo info;
    if (isLocal) {
        info = sqlManager.getDataCenterInfo(datacenter_name, now_time);
    } else {
        info = sqlManager.getDataCenterInfo(datacenter_name, now_time - history_delay);
    }
    if (info.hosts.size() == 0 && isLocal) {
        return true;
    } else if (info.hosts.size() == 0 && !isLocal) {
        return false;
    }
    boolean issuit = true;
    for (VmRegistry vm : customer.getVms()) {
        boolean issuitone = false;
        for (int i = 0; i < info.hosts.size(); i++) {
            HostInfo hinfo = info.hosts.get(i);
            if (hinfo.ram >= vm.getRam() && hinfo.storage >= vm.getSize() && hinfo.bw >= vm.getBw()
                    && hinfo.free_mips >= vm.getMips()) {
                hinfo.ram -= vm.getRam();
                hinfo.storage -= vm.getSize();
                hinfo.bw -= vm.getBw();
                hinfo.free_mips -= vm.getMips();
                issuitone = true;
                break;
            }
        }
        if (!issuitone) {
            issuit = false;
            break;
        }
    }
    return issuit;
}
```

首先根据 SqlManager 来获取数据中心的主机历史信息，注意获取的时候分为本地和其他数据中心，如果获取其他数据中心则就只能够获取一定延迟之前的主机的历史信息，如果信息都没有，那么就说明模拟刚开始，默认放到本地数据中心。判断是否合适时，主要需要看的是对于所有的虚拟机，数据中心中是否有合适的主机能放置该虚拟机，该主机是否能放置的标准是是否有足够的内存，是否有足够的存储空间，是否有足够的宽带，是否有足够的计算能力，如果都有就算合适，如果全部虚拟机都合适了，那么就没问题了，这个客户需求就可以放置在这个数据中心。

3 实验分析

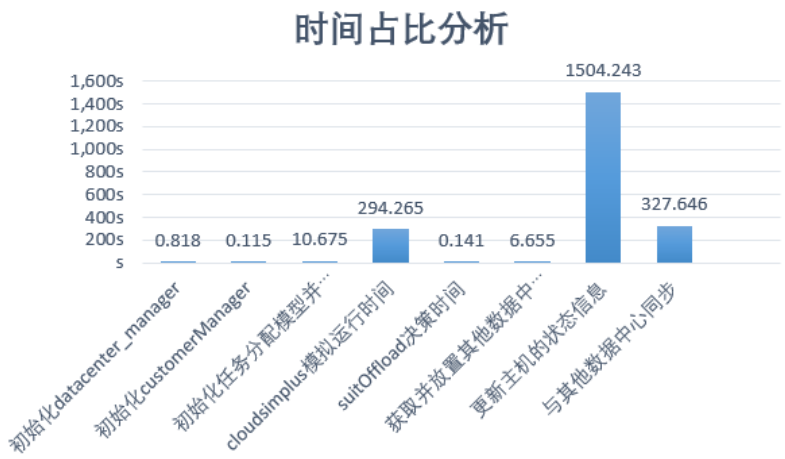
3.1 评价标准

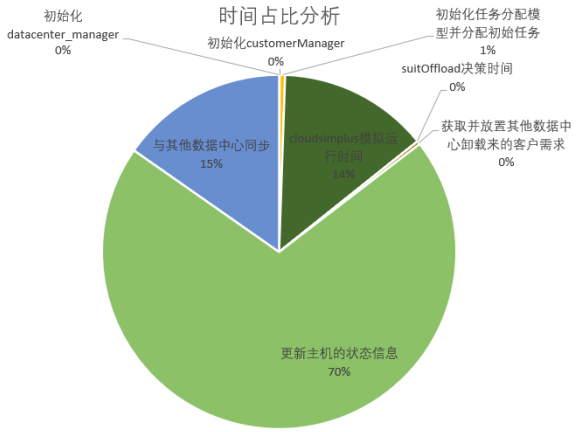
本实验需要达到多服务器共同运行同一个模拟器的模拟效果，所以一方面针对程序的运行时间进行分析，一方面对模拟的效果进行分析，各个客户需求分配策略需要达到预期的效果。

3.2 实验结果与分析

程序运行时间占比分析

这里用 2 个服务器来模拟两个数据中心。服务器的配置为 2 核 4G，每个核的配置为：Intel(R) Xeon(R) Gold 6161 CPU @ 2.20GHz。然后每个数据中心有 10000 台主机，每个数据中心拥有 1000 个客户，每个客户都需要创建 10 台虚拟机，有 10 个任务在上面执行。使用的是循环适配的方法来决策任务应该流向哪个数据中心。模拟 30s 所花费的时间是 2132.987s，即 35.55 分钟。最后得到的时间占比分析如下：





可以看出主要的时间都花费在了更新主机的状态信息上，总共花费了 1504 秒，占比 70%。这主要是因为设置的模拟间隔是 0.1 秒，所以每 0.1s 就需要把 1 万个主机的状态信息都同步到数据库中去，数据提交所需要的时间多。再然后多的时间就是和其他数据中心同步用的时间花费了 327 秒，占比 15%，因为每模拟一段 0.1 秒就需要同步其他数据中心，需要等待其将模拟的时间发送过来。再然后时间占比第三的才是 cloudsim plus 模拟运行的时间，花费了 294 秒，这也就说明了真实用在模拟数据中心的行为的时间反而是比较少的。其他的一些时间也就更少了，基本可以忽略不计。

模拟效果分析

randomOffload（随机分配）

这里依据是使用 2 个数据中心来进行模拟，其中一个数据中心 c1 有 2 个客户需求，一个在一开始 0s 的时候提交上去，一个在 10s 的时候提交上去；另一个数据中心 c2 有 5 个客户需求，一个在一开始 0s 的时候提交上去，一个在 10s 的时候提交上去，剩余三个在 15s 的时候全部提交上去。各个数据中心的主机都管够。最后运行的结果如下：

c1 数据中心：

从c2来的c2_0类第0个

Cloudlet ID	Status	DC ID	Host ID	Host PEs CPU cores	VM ID	VM PEs CPU cores	CloudletLen MI	FinishedLen MI	CloudletPEs CPU cores	StartTime Seconds	FinishTime Seconds	ExecTime Seconds
0	SUCCESS	1	0	40	0	4	5000	5000	4	0	10	10
1	SUCCESS	1	1	40	1	4	5000	5000	4	0	10	10
2	SUCCESS	1	2	40	2	4	5000	5000	4	0	10	10
3	SUCCESS	1	3	40	3	4	5000	5000	4	0	10	10
4	SUCCESS	1	4	40	4	4	5000	5000	4	0	10	10

从c2来的c2_1类第0个

Cloudlet ID	Status	DC ID	Host ID	Host PEs CPU cores	VM ID	VM PEs CPU cores	CloudletLen MI	FinishedLen MI	CloudletPEs CPU cores	StartTime Seconds	FinishTime Seconds	ExecTime Seconds
0	SUCCESS	1	0	40	0	4	5000	5000	2	11	15	5
1	SUCCESS	1	1	40	1	4	5000	5000	2	11	15	5
2	SUCCESS	1	2	40	2	4	5000	5000	2	11	15	5
3	SUCCESS	1	3	40	3	4	5000	5000	2	11	15	5
4	SUCCESS	1	4	40	4	4	5000	5000	2	11	15	5

从c2来的c2_2类第0个

Cloudlet ID	Status	DC ID	Host ID	Host PEs CPU cores	VM ID	VM PEs CPU cores	CloudletLen MI	FinishedLen MI	CloudletPEs CPU cores	StartTime Seconds	FinishTime Seconds	ExecTime Seconds
0	SUCCESS	1	5	40	0	4	5000	5000	4	16	25	10
1	SUCCESS	1	6	40	1	4	5000	5000	4	16	25	10
2	SUCCESS	1	7	40	2	4	5000	5000	4	16	25	10
3	SUCCESS	1	8	40	3	4	5000	5000	4	16	25	10
4	SUCCESS	1	9	40	4	4	5000	5000	4	16	25	10

c2 数据中心：

从c1来的c1_0类第0个

Cloudlet ID	Status	DC ID	Host ID	Host PEs CPU cores	VM ID	VM PEs CPU cores	CloudletLen MI	FinishedLen MI	CloudletPEs CPU cores	StartTime Seconds	FinishTime Seconds	ExecTime Seconds
0	SUCCESS	1	0	40	0	4	5000	5000	4	0	10	10
1	SUCCESS	1	1	40	1	4	5000	5000	4	0	10	10
2	SUCCESS	1	2	40	2	4	5000	5000	4	0	10	10
3	SUCCESS	1	3	40	3	4	5000	5000	4	0	10	10
4	SUCCESS	1	4	40	4	4	5000	5000	4	0	10	10

从c1来的c1_1类第0个

Cloudlet ID	Status	DC ID	Host ID	Host PEs CPU cores	VM ID	VM PEs CPU cores	CloudletLen MI	FinishedLen MI	CloudletPEs CPU cores	StartTime Seconds	FinishTime Seconds	ExecTime Seconds
0	SUCCESS	1	5	40	0	4	5000	5000	2	10	15	5
1	SUCCESS	1	6	40	1	4	5000	5000	2	10	15	5
2	SUCCESS	1	7	40	2	4	5000	5000	2	10	15	5
3	SUCCESS	1	8	40	3	4	5000	5000	2	10	15	5
4	SUCCESS	1	9	40	4	4	5000	5000	2	10	15	5
5	SUCCESS	1	5	40	0	4	5000	5000	2	10	24	14

c2_2类第1个

Cloudlet ID	Status	DC ID	Host ID	Host PEs CPU cores	VM ID	VM PEs CPU cores	CloudletLen MI	FinishedLen MI	CloudletPEs CPU cores	StartTime Seconds	FinishTime Seconds	ExecTime Seconds
0	SUCCESS	1	0	40	0	4	5000	5000	4	15	25	10
1	SUCCESS	1	1	40	1	4	5000	5000	4	15	25	10
2	SUCCESS	1	2	40	2	4	5000	5000	4	15	25	10
3	SUCCESS	1	3	40	3	4	5000	5000	4	15	25	10
4	SUCCESS	1	4	40	4	4	5000	5000	4	15	25	10

c2_2类第2个

Cloudlet ID	Status	DC ID	Host ID	Host PEs CPU cores	VM ID	VM PEs CPU cores	CloudletLen MI	FinishedLen MI	CloudletPEs CPU cores	StartTime Seconds	FinishTime Seconds	ExecTime Seconds
0	SUCCESS	1	10	40	0	4	5000	5000	4	15	25	10
1	SUCCESS	1	11	40	1	4	5000	5000	4	15	25	10
2	SUCCESS	1	12	40	2	4	5000	5000	4	15	25	10
3	SUCCESS	1	13	40	3	4	5000	5000	4	15	25	10
4	SUCCESS	1	14	40	4	4	5000	5000	4	15	25	10

可以看到 c1 数据中心的 2 个客户需求都被随机卸载到了 c2 数据中心，开始 c2 运行的时间分别是 0s 和 10s。而 c2 数据中心的 5 个客户需求中也有 3 个被随机卸载到了 c1，在 c1 开始运行的时间为 0s、11s 和 16s。如此结果说明随机卸载策略可以正常运行。

cheapesOffload（资源花费最低分配）

这里的实验配置与上一个基本相同，但是设置了 c1 数据中心的内存价格是 0 元，设置 c2 数据中心的宽带价格是 0 元。然后设置 c1 中在 0s 提交的客户需求中 vm 的宽带要求是 10000，内存要求是 2000，而在 10s 提交的客户需求中 vm 的宽带要求是 1000，内存要求是 20000，而其他基本相同，所以理论上 0s 的客户需求应该放置在 c2 数据中心，10s 的客户需求应该放置在 c1 数据中心。运行结果如下：

c1 数据中心：

```
                                                    c1_1类第0个
|Cloudlet|Status |DC|Host|Host PEs |VM|VM PEs   |CloudletLen|FinishedLen|CloudletPEs|StartTime|FinishTime|ExecTime
|--------|-------|--|----|---------|--|---------|-----------|-----------|-----------|---------|----------|--------
|      ID|       |ID|  ID|CPU cores|ID|CPU cores|         MI|         MI|  CPU cores|  Seconds|   Seconds| Seconds
|       0|SUCCESS| 1|   0|       40| 0|        4|       5000|       5000|          2|       10|        15|       5
|       1|SUCCESS| 1|   1|       40| 1|        4|       5000|       5000|          2|       10|        15|       5
|       2|SUCCESS| 1|   2|       40| 2|        4|       5000|       5000|          2|       10|        15|       5
|       3|SUCCESS| 1|   3|       40| 3|        4|       5000|       5000|          2|       10|        15|       5
|       4|SUCCESS| 1|   4|       40| 4|        4|       5000|       5000|          2|       10|        15|       5

Vm 0 costs ($) for     5.52 execution seconds - CPU:     0.04$ RAM:     0.00$ Storage:     0.00$ BW:   100.00$ Total:   100.04$
Vm 1 costs ($) for     5.52 execution seconds - CPU:     0.04$ RAM:     0.00$ Storage:     0.00$ BW:   100.00$ Total:   100.04$
Vm 2 costs ($) for     5.52 execution seconds - CPU:     0.04$ RAM:     0.00$ Storage:     0.00$ BW:   100.00$ Total:   100.04$
Vm 3 costs ($) for     5.52 execution seconds - CPU:     0.04$ RAM:     0.00$ Storage:     0.00$ BW:   100.00$ Total:   100.04$
Vm 4 costs ($) for     5.52 execution seconds - CPU:     0.04$ RAM:     0.00$ Storage:     0.00$ BW:   100.00$ Total:   100.04$
c1_1类第0个 total cost ($) for   5 VMs in   5 created VMs :     0.22$          0.00$            0.00$       500.00$          500.22$
```

c2 数据中心：

```
                                                    从c1来的c1_0类第0个
|Cloudlet|Status |DC|Host|Host PEs |VM|VM PEs   |CloudletLen|FinishedLen|CloudletPEs|StartTime|FinishTime|ExecTime
|--------|-------|--|----|---------|--|---------|-----------|-----------|-----------|---------|----------|--------
|      ID|       |ID|  ID|CPU cores|ID|CPU cores|         MI|         MI|  CPU cores|  Seconds|   Seconds| Seconds
|       0|SUCCESS| 1|   5|       40| 0|        4|       5000|       5000|          4|        0|        10|      10
|       1|SUCCESS| 1|   6|       40| 1|        4|       5000|       5000|          4|        0|        10|      10
|       2|SUCCESS| 1|   7|       40| 2|        4|       5000|       5000|          4|        0|        10|      10
|       3|SUCCESS| 1|   8|       40| 3|        4|       5000|       5000|          4|        0|        10|      10
|       4|SUCCESS| 1|   9|       40| 4|        4|       5000|       5000|          4|        0|        10|      10

Vm 0 costs ($) for    10.50 execution seconds - CPU:     0.08$ RAM:   100.00$ Storage:    25.00$ BW:     0.00$ Total:   125.08$
Vm 1 costs ($) for    10.50 execution seconds - CPU:     0.08$ RAM:   100.00$ Storage:    25.00$ BW:     0.00$ Total:   125.08$
Vm 2 costs ($) for    10.50 execution seconds - CPU:     0.08$ RAM:   100.00$ Storage:    25.00$ BW:     0.00$ Total:   125.08$
Vm 3 costs ($) for    10.50 execution seconds - CPU:     0.08$ RAM:   100.00$ Storage:    25.00$ BW:     0.00$ Total:   125.08$
Vm 4 costs ($) for    10.50 execution seconds - CPU:     0.08$ RAM:   100.00$ Storage:    25.00$ BW:     0.00$ Total:   125.08$
从c1来的c1_0类第0个 total cost ($) for   5 VMs in   5 created VMs :     0.42$        500.00$          125.00$          0.00$          625.42$
```

这结果与预想中的相同，一个花费了 500$，一个花费了 625$，说明该卸载策略可以正常运行

suitOffload（循环适配）

这里依旧是 2 个数据中心，我们设置 c1 的数据中心的主机只有 5 台，每台主机 4 个 pe 核，每个核的计算能力是 1100mips，而在 0 时刻提交的客户需求将会创建 5 个 4 核计算能力需要为 1000mips 的主机，并且任务会运行 16s，即 vm 会存在 16s，所以当 c1 的另一份客户需求在 10s 的时候提交的时候就会被转交给 c2 数据中心，而我们保证 c2 数据中心有足够的计算能力。模拟结果如下：

c1 数据中心：

c1_0类第0个

Cloudlet ID	Status	DC ID	Host ID	Host PEs CPU cores	VM ID	VM PEs CPU cores	CloudletLen MI	FinishedLen MI	CloudletPEs CPU cores	StartTime Seconds	FinishTime Seconds	ExecTime Seconds
0	SUCCESS	1	0	4	0	4	16000	16000	4	0	16	16
1	SUCCESS	1	1	4	1	4	16000	16000	4	0	16	16
2	SUCCESS	1	2	4	2	4	16000	16000	4	0	16	16
3	SUCCESS	1	3	4	3	4	16000	16000	4	0	16	16
4	SUCCESS	1	4	4	4	4	16000	16000	4	0	16	16

c2 数据中心：

从c1来的c1_1类第0个

Cloudlet ID	Status	DC ID	Host ID	Host PEs CPU cores	VM ID	VM PEs CPU cores	CloudletLen MI	FinishedLen MI	CloudletPEs CPU cores	StartTime Seconds	FinishTime Seconds	ExecTime Seconds
0	SUCCESS	1	10	40	0	4	5000	5000	2	10	15	5
1	SUCCESS	1	11	40	1	4	5000	5000	2	10	15	5
2	SUCCESS	1	12	40	2	4	5000	5000	2	10	15	5
3	SUCCESS	1	13	40	3	4	5000	5000	2	10	15	5
4	SUCCESS	1	14	40	4	4	5000	5000	2	10	15	5

c2_2类第0个

Cloudlet ID	Status	DC ID	Host ID	Host PEs CPU cores	VM ID	VM PEs CPU cores	CloudletLen MI	FinishedLen MI	CloudletPEs CPU cores	StartTime Seconds	FinishTime Seconds	ExecTime Seconds
0	SUCCESS	1	0	40	0	4	5000	5000	4	15	25	10
1	SUCCESS	1	1	40	1	4	5000	5000	4	15	25	10

可以看到第 10s 在 c1 提交的被转交给了 c2。查看主机的历史记录：

```
MariaDB [cloudsimPlus]> select * from c1_history;
```

id	host_id	run_time	ram	bw	storage	free_pes	free_mips
1	0	0	1000000	100000	100000	4	4400
2	1	0	1000000	100000	100000	4	4400
3	2	0	1000000	100000	100000	4	4400
4	3	0	1000000	100000	100000	4	4400
5	4	0	1000000	100000	100000	4	4400
6	0	1.1	998000	90000	99500	0	400
7	1	1.1	998000	90000	99500	0	400
8	2	1.1	998000	90000	99500	0	400
9	3	1.1	998000	90000	99500	0	400
10	4	1.1	998000	90000	99500	0	400
11	0	2.1	998000	90000	99500	0	400
12	1	2.1	998000	90000	99500	0	400
13	2	2.1	998000	90000	99500	0	400
14	3	2.1	998000	90000	99500	0	400

……

78	2	16.1	998000	90000	99500	0	400
79	3	16.1	998000	90000	99500	0	400
80	4	16.1	998000	90000	99500	0	400
81	0	17.11	999000	100000	100000	4	4400
82	1	17.11	999000	100000	100000	4	4400
83	2	17.11	999000	100000	100000	4	4400
84	3	17.11	999000	100000	100000	4	4400
85	4	17.11	999000	100000	100000	4	4400
86	0	18.11	999000	100000	100000	4	4400
87	1	18.11	999000	100000	100000	4	4400
88	2	18.11	999000	100000	100000	4	4400
89	3	18.11	999000	100000	100000	4	4400
90	4	18.11	999000	100000	100000	4	4400
91	0	19.11	999000	100000	100000	4	4400
92	1	19.11	999000	100000	100000	4	4400
93	2	19.11	999000	100000	100000	4	4400
94	3	19.11	999000	100000	100000	4	4400

可以发现也和之前预料的一样，在 16s 之前 CPU 一直处于被占用状态，每个主机可用的 mips 只有 400，无法让在 10s 提交的客户需求放置上去。

4 结论与展望

本次模拟器的设计完成过程中我学到了很多。首先是一开始的时候和师兄他们一起看论文讨论选题，自己看论文的从看的一头雾水到逐渐能够参得论文作者所写的主要中心思想，能够把论文读薄。然后是在写模拟器的过程也是一个逐渐学习的过程，首先是先深入细致地学习了 clousim plus 这个模拟器，然后就是学习 java，原本是完全没有写过 java 代码的，自己在整个过程中都是边学边写，目前已经对 java 的基本语法基本熟练了，而且在写的过程中也是学习了 java 的网络编程，java 的文件处理，java 的 maven 加载其他库，java 操作数据库等等主题的内容，使得自己对 java 编程也初上门道了。

本次模拟器最后也是顺利地实现了我预想中的所有功能，能够在多个服务器上顺利执行同一个模拟器来共同完成模拟，拓宽了模拟器的能力上限。但是正如程序运行时间分析中所提到的，太多的时间浪费在了与数据库的交互上面，所以后面在针对研究的算法进行具体的实验的时候需要有针对性地优化数据库的操作，例如不需要每一次的主机状态信息都提交给数据库，以此来将大规模模拟的时间降下来。而且总体看来为了提高大规模的能力而将模拟器分散到多个服务器还需要更多的精心调磨，因为一旦是多个服务器，服务器之间的信息同步就只能通过数据库或者网络连接这种形式，而不能像在同一个服务器里的通过内存来进行快速的信息交流，如何把时间主要花费在模拟上而不是在各个服务器之间的信息交流上是提高模拟器速度的关键，这也是设计过程中特别需要注意的

参考文献

[1] Bambrik, I. A Survey on Cloud Computing Simulation and Modeling. SN COMPUT. SCI. 1, 249 (2020). https://doi.org/10.1007/s42979-020-00273-1

[2] N. Naik and M. A. Mehta, "Comprehensive and Comparative Study of Cloud Simulators," 2018 IEEE Punecon, 2018, pp. 1-7, doi: 10.1109/PUNECON.2018.8745422.

[3] Elahi, B.M., Malik, A.W., Rahman, A.U., & Khan, M.A. (2020). Toward scalable cloud data center simulation using high‐level architecture. Software: Practice and Experience, 50, 827 - 843.

[4] Monika Gill, Dinesh Singh, A comprehensive study of simulation frameworks and research directions in fog computing ,Computer Science Review, Volume 40,2021,100391,ISSN 1574-0137,https://doi.org/10.1016/j.cosrev.2021.100391.

[5] M. C. Silva Filho, R. L. Oliveira, C. C. Monteiro, P. R. M. Inácio and M. M. Freire, "CloudSim Plus: A cloud computing simulation framework pursuing software engineering principles for improved modularity, extensibility and correctness," 2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM), 2017, pp. 400-406, doi: 10.23919/INM.2017.7987304.

[6] Wang, Zi-Jia et al. “Dynamic Group Learning Distributed Particle Swarm Optimization for Large-Scale Optimization and Its Application in Cloud Workflow Scheduling.” IEEE Transactions on Cybernetics 50 (2020): 2715-2729.

[7] Zhong, Xiaoxiong et al. “POTAM: A Parallel Optimal Task Allocation Mechanism for Large-Scale Delay Sensitive Mobile Edge Computing.” IEEE Transactions on Communications 70 (2022): 2499-2517.

[8] Huang, Liang et al. “Deep Reinforcement Learning for Online Computation Offloading in Wireless Powered Mobile-Edge Computing Networks.” IEEE Transactions on Mobile Computing 19 (2018): 2581-2593.

[9] Zou, Junfeng et al. “A3C-DO: A Regional Resource Scheduling Framework Based on Deep Reinforcement Learning in Edge Scenario.” IEEE Transactions on Computers 70 (2021): 228-239.

[10] Wang, Yong et al. “Joint Deployment and Task Scheduling Optimization for Large-Scale Mobile Users in Multi-UAV-Enabled Mobile Edge Computing.” IEEE Transactions on Cybernetics 50 (2020): 3984-3997.

[11] Gao, Zhen et al. “Large-Scale Computation Offloading Using a Multi-Agent Reinforcement Learning in Heterogeneous Multi-access Edge Computing.” IEEE Transactions on Mobile Computing (2022): n. pag.

[12] Hong, Yuncong et al. “Distributed Job Dispatching in Edge Computing Networks With Random Transmission Latency: A Low-Complexity POMDP Approach.” IEEE Internet of Things Journal 9 (2021): 4152-4167.