分布式人脸识别系统演示

背景

什么是分布式系统

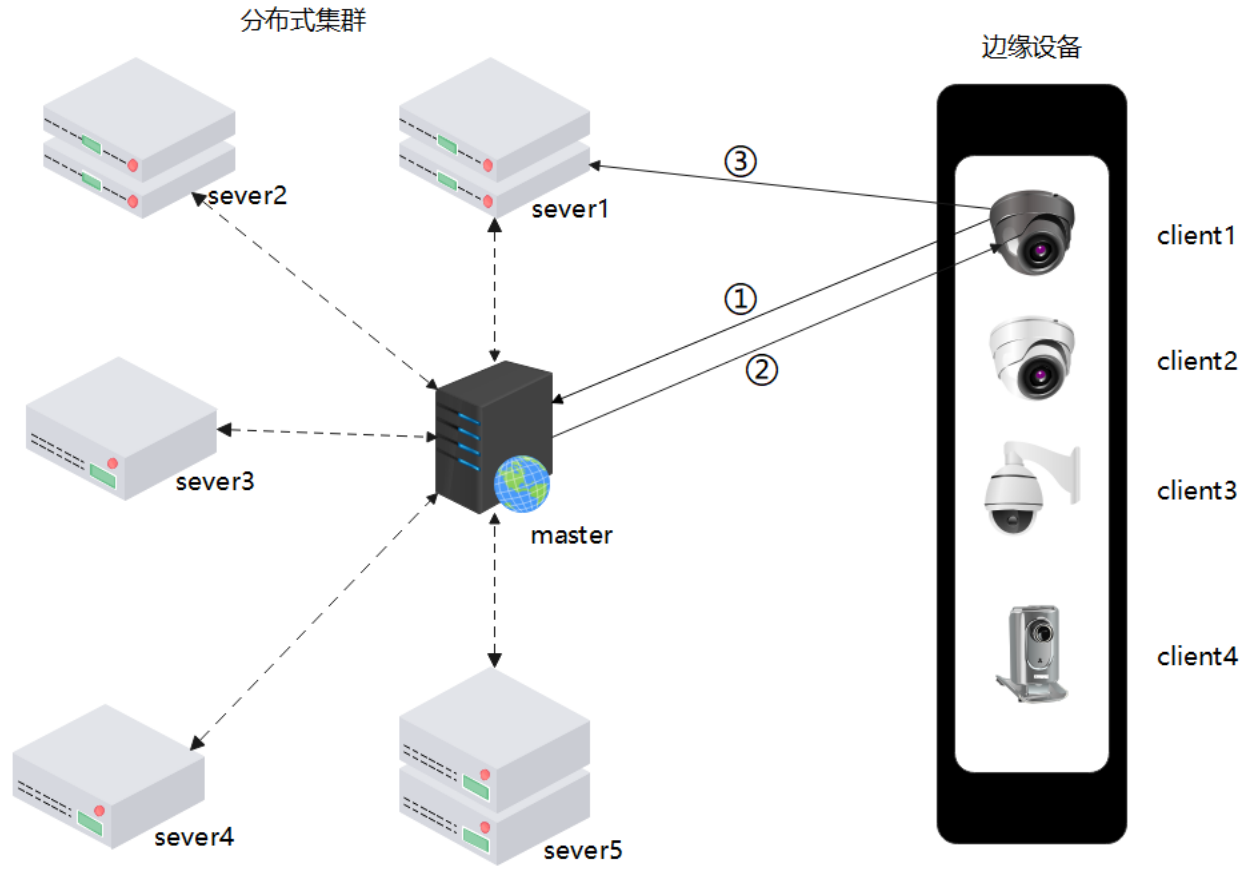
分布式系统是由一组通过网络进行通信、为了完成共同的任务而协调工作的计算机节点组成的系统。分布式系统的出现是为了用廉价的、普通的机器完成单个计算机无法完成的计算、存储任务。其目的是利用更多的机器，处理更多的数据。当单个节点的处理能力无法满足日益增长的计算、存储任务的时候，且硬件的提升（加内存、加磁盘、使用更好的CPU）高昂到得不偿失的时候，应用程序也不能进一步优化的时候，我们才需要考虑分布式系统

我们能够用分布式系统来做什么

一个服务器的计算能力是存在波动的，会有比如温度等其他影响因素，这个时候为了能够使得边缘设备的任务能够得到及时的处理，需要让边缘设备知道有自己需要提交任务到哪个服务器节点，用这种方式来实现多个服务器节点的协同计算与边缘设备的任务调度

主节点进行资源监控与节点调度，协调多个服务器节点并行处理边缘设备提交的任务

分布式图像处理系统结构



这个分布式系统主要由三种设备：`master节点`， `server节点`和`client设备`。

`master节点`是主节点，本身不参与计算，用来统筹和协调server节点和clien设备之间的任务分配和资源请求，管理所有server节点的资源。

`server节点`负责计算服务，在和master节点建立rpc连接，接受master节点的管理。

`client程序`是提交图像信息和计算任务请求的设备，它只知道master的地址并和master进行直接通信。在需要提交计算任务时，会向master发送rpc请求，master节点会返回合适的server节点的网络地址信息给client设备，接下来client会再向server节点提交计算任务。

`rpc协议`在master、server、client之间建立信道，传输图像信息或者通知信息

具体演示运行

master节点的工作

- `master`节点需要和`server`节点进行握手，管理他们的信息(CPU占用率等资源统计信息)。同时，master提供了`调度算法`，在client向master发来`rpc请求`之后，master会根据client的任务占用资源的多少、每一个server的资源统计信息来决定让哪一个server去执行client的任务，然后master会将这个server的`ip:port`发送给client，client成功接收返回之后继而向该server提交任务。

- master在端口中暴露5种服务：`Join`、Leave、`PushState`、`RouteGuide`、getQ
  - `Join`：设备启动时，server通过rpc协议与master建立信道连接，发送server的ip:port给master，从而加入服务器集群`cluster`
  - `RouteGuide`：clietn向master发起rpc请求并成功建立信道之后，会调用master的`RouteGuide`服务，向master询问server的ip:port。master会通过client提交的rpc请求信息和server统计信息进行调度决策，之后返回server的ip:port给client
  - `PushState`：在设备启动并且client加入到master的服务器集群之后，每隔一秒，server就调用master的`PushState`服务，`PushState`服务负责让master从rpc请求中获取server的最新资源统计信息，更新自己本地存储的server信息。

代码展示

```
from concurrent import futures
import time
import grpc
import master_pb2
import master_pb2_grpc
from multiprocessing import Process, Manager
import queue
import json
from node import Node
import sys



def recordHeartBeat(q, file):
    with open(file, 'w') as f:
        while True:
            data = q.get()
            print("writing")
            data_str = json.dumps(data)
            f.write(data_str + "\n")
            f.flush()

class Master():
    def __init__(self):
        self.nodes = {}
        self.nodes_weights = {}
        self.nodes_vals = {}
        self.last_index = 0
        self.node_ids = []
        self.recodQ = Manager().Queue()


    def getQueue(self):
        return self.recodQ

    def generateID(self, ip, port):
        return len(self.node_ids)

    def join(self, ip, port, weight):
        id = self.generateID(ip, port)
```

```
        if id in self.nodes:
            return False
        self.nodes[id] = Node(ip, port)
        self.nodes_weights[id] = weight
        self.nodes_vals[id] = weight
        self.node_ids.append(id)
        return id, True
    def leave(self, id):
        if id not in self.nodes:
            return False
        self.nodes.pop(id)
        return True

    def updateInfo(self, id, info):
        if id not in self.nodes:
            print("invalid ")
            return False
        self.recodQ.put(info)
        self.nodes[id].update(info)
        return True

    def scheduler(self, task_type, task_sz):
        next_index = -1
        wt = 0
        for id in self.nodes_vals.keys():
            if self.nodes_vals[id] > wt:
                next_index = id
                wt = self.nodes_vals[id]
        if next_index == -1:
            for id in self.nodes_vals.keys():
                self.nodes_vals[id] = self.nodes_weights[id]
            return self.scheduler(task_type, task_sz)
        self.nodes_vals[next_index] -= 1

        index = self.node_ids[next_index]
        ip = self.nodes[index].getIp()
        port = self.nodes[index].getPort()
        print(ip, port)
        return ip, port

    def getNodesNum(self):
        return len(self.nodes)
class MasterService(master_pb2_grpc.MasterServicer):
    ms = Master()
    # worker node join to the cluster
    # send the server port and ip to master
    def Join(self, request, context):
        ip = request.ip
        port = request.port
        weight = request.weight
        print("new join ",ip, port)
        id, res = self.ms.join(ip, port, weight)
        print("join id is ", id)
        return master_pb2.JoinReply(id = id,success = res)

    def Leave(self, request, context):
        id = request.id
        res = self.ms.leave(id)
        return master_pb2.LeaveReply(res)

    def PushState(self, request, context):
```

```
        id = request.id
        print("heartbeat id is ", id)
        info = {}
        info['id'] = request.id
        info['utx'] = request.utx
        info['cpu_usage'] = request.cpu_usage
        info['task_id'] = request.task_id
        info['io_perf'] = request.io_perf
        info['pro_perf'] = request.pro_perf
        info['task_st'] = request.task_st
        info['task_ed'] = request.task_ed

        # if request.task_st != 0:
            # print(request.task_id, request.io_perf, request.pro_perf, request.task_st, request.task_ed)
        # else:
            # print(id, request.utx ,request.cpu_usage)
        # print(id, request.io_perf, request.pro_perf)
        self.ms.updateInfo(id, info)
        return master_pb2.SateReply()

    # send target server info to client
    def RouteGuide(self, request, context):
        if self.ms.getNodesNum() == 0:
            return master_pb2.RouteReply(ip = "", port = 555, thread_num = 2)
        print("nodes num = ", self.ms.getNodesNum())
        task_type = request.task_type
        task_sz = request.task_sz
        print("cur task is", task_type, task_sz)
        next_ip, next_port = self.ms.scheduler(task_type, task_sz)
        print("schedle ", next_ip, next_port)
        return master_pb2.RouteReply(ip = next_ip, port = next_port)

    def getQ(self):
        return self.ms.getQueue()

if __name__ == "__main__":
    server = grpc.server(futures.ThreadPoolExecutor(max_workers=2))
    service = MasterService()
    master_pb2_grpc.add_MasterServicer_to_server(service, server)
    #bind a port
    server.add_insecure_port('[::]:50051')
    server.start()
    recordP = Process(target = recordHeartBeat, args = (service.getQ(), "heartbeat"))
    recordP.start()
    try:
        while True:
            time.sleep(60*60*24)
    except KeyboardInterrupt:
        server.stop(0)
```

server节点的工作

- 在设备启动时，`server节点`需要向`master节点`发送`rpc请求`，成功之后会调用master节点的`Join服务`，将自己的网络地址传递给master，从而加入服务器集群。
- 之后为了向master节点报告自己的资源统计信息，server节点每隔一秒就会调用master的`PushState`服务，更新master中自己的资源统计信息。

代码展示

```
import asyncio
import grpc
import stream_video_pb2
import stream_video_pb2_grpc

import master_pb2
import master_pb2_grpc
import random
import numpy as np
from cv2 import cv2 as cv

def get_rand():
    return random.randint(0, 55555)

class VideoStreamServicer(stream_video_pb2_grpc.VideoStreamServicer):
    def __init__(self):
        self.loadModel()
    def loadModel(self):
        self.faceCascade = cv.CascadeClassifier("Image/haarcascade_frontalface_default.xml")

    def SendFream(self, request, context):
        nparr = np.frombuffer(request.fream, np.uint8)
        img = cv.imdecode(nparr, cv.IMREAD_COLOR)
        imgResise = cv.resize(img, (500,400))
        imgGray = cv.cvtColor(imgResise, cv.COLOR_BGR2GRAY)
        faces = self.faceCascade.detectMultiScale(imgResise, 1.1, 4)
        for (x, y, w, h) in faces:
            cv.rectangle(imgResise, (x, y), (x + w, y + h), (255, 0, 0), 1)

        return stream_video_pb2.Result()

    def PushStream(self, request_iterator, context):
        for fream in request_iterator:
            print(fream.size)
        return stream_video_pb2.Result()

    async def PushAndPull(self, request_iterator, unused_context):
        async for req in request_iterator:
            nparr = np.frombuffer(req.fream, np.uint8)
            img = cv.imdecode(nparr, cv.IMREAD_COLOR)
            imgResise = cv.resize(img, (500,400))
            imgGray = cv.cvtColor(imgResise, cv.COLOR_BGR2GRAY)
            faces = self.faceCascade.detectMultiScale(imgResise, 1.1, 4)
            for (x, y, w, h) in faces:
                cv.rectangle(imgResise, (x, y), (x + w, y + h), (255, 0, 0), 1)
            img_encode = cv.imencode('.jpg', imgResise)[1].tobytes()
            data_encode = np.array(img_encode)
            str_encode = data_encode.tobytes()
            yield stream_video_pb2.Fream(size = len(str_encode), fream = str_encode)

    def FaceDetection(self, request_iterator, context):
        ri = get_rand()
        for req in request_iterator:
            nparr = np.frombuffer(req.frame, np.uint8)
            img = cv.imdecode(nparr, cv.IMREAD_COLOR)
            imgResise = cv.resize(img, (500,400))
            imgGray = cv.cvtColor(imgResise, cv.COLOR_BGR2GRAY)
```

```
            faces = self.faceCascade.detectMultiScale(imgResise, 1.1, 4)
            for (x, y, w, h) in faces:
                cv.rectangle(imgResise, (x, y), (x + w, y + h), (255, 0, 0), 1)
            cv.imshow('FaceDetection{}'.format(ri), imgResise)
            cv.waitKey(1)
        cv.destroyAllWindows()
        return stream_video_pb2.Result()
    def Canny(self, request_iterator, context):
        for req in request_iterator:
            nparr = np.frombuffer(req.frame, np.uint8)
            img = cv.imdecode(nparr, cv.IMREAD_COLOR)
            blur = cv.GaussianBlur(img, (3, 3), 0)
            canny = cv.Canny(blur, 50, 150)
            cv.imshow('Canny', canny)
            cv.waitKey(1)
        cv.destroyAllWindows()
        return stream_video_pb2.Result()

async def push_state(stub, id):
    await stub.PushState(master_pb2.StateRequest(id = id, cpu_usage = 50))

async def join(ip, port, stub, wt):
    res = await stub.Join(master_pb2.JoinRequest(ip = ip,port = port, weight = wt))
    return res.id

# 1. join to the cluster and send local ip and port to the master
async def heartBeatTimer(ip, port, weight):
    # insecure_channel build a channel with master
    async with grpc.aio.insecure_channel('169.254.144.84:50051') as channel:
        stub = master_pb2_grpc.MasterStub(channel)
        # rpc, join to the cluster
        id = await join(ip, port, stub, weight)
    while True:
        await asyncio.sleep(1)
        # heart message
        async with grpc.aio.insecure_channel('169.254.144.84:50051') as channel:
            stub = master_pb2_grpc.MasterStub(channel)
            await push_state(stub, id)



async def serve() -> None:
    server = grpc.aio.server()
    stream_video_pb2_grpc.add_VideoStreamServicer_to_server(
                                    VideoStreamServicer(), server)
    server.add_insecure_port('[::]:50050')
    await server.start()
    await server.wait_for_termination()



async def main(ip, port, weight):
    await asyncio.gather(
        heartBeatTimer(ip, port, weight),
        serve()
    )

if __name__ == "__main__":
    # server = grpc.server(futures.ThreadPoolExecutor(max_workers=2))
    # service = StreamService()
    # service.loadModel()
```

```
    # stream_video_pb2_grpc.add_VideoStreamServicer_to_server(service, server)
    # server.add_insecure_port('[::]:50051')
    # server.start()

    # try:
    #     while True:
    #         time.sleep(60*60*24)
    # except KeyboardInterrupt:
    #     server.stop(0)
    # asyncio.get_event_loop().run_until_complete(serve())

    # server local ip and port
    asyncio.run(main("169.254.10.121", 50050, 2))
```

client程序的工作

- client与master建立联系，继而请求master的`RouteGuide`服务。在收到master返回的server的`ip:port`之后，client向server请求建立联系，提交任务。
- client 可以通过在虚拟机上产生多个进程来去模拟不同的client。

具体演示流程

1. 首先在我的电脑上开启master节点，master节点运行mster.py文件
2. 使用三台树梅派当作server节点上面运行stream_video_server.py文件，（server必须先知道master的ip:port，然后手动在server的代码中进行修改，所以必须先启动master再启动server）
3. 在另一台电脑上运行多个stream_video_client文件模拟client【为了简化情况，我们使用本地MP4文件替代摄像头的图像输入】
4. 注意client里面读取mp4文件的路径也是硬编码的，需要修改为你对应的路径

注意事项

配置python环境

```
numpy
scipy
matplotlib
grpcio
protobuf
grpcio-tools
asyncio
aiohttp
opencv-contrib-python
```

- 备注：
  - 必须要按照以上顺序安装模块，主要是opencv这个模块，如果不按照这个顺序安装模块的话，电脑会被占据大量CPU 和内存，然后崩溃，并且不能恢复（已经重装了一台双系统 两台虚拟机 一台云服务器）
  - python版本要大于或等于3.7

配置网络环境

- master和server和client必须关闭防火墙以防万一

其他

- win10上不能运行master，win10对fork进程的限制会导致报错
- 只能在虚拟机或者mac或者ubantu上运行master
- 在虚拟机上运行mster的话，要设置网络状态为桥接模式，从而获取一个公网的ip地址，写到server和client代码中【但通常我的虚拟机改为桥接之后连不上网】
- 校园网必须输入账号密码，不然是内网模式，不能与其他主机ping通