ERM V1.0

欢迎来到 ERM V1.0。

项目描述

一个面向边缘计算的多用户计算资源分配模型：

基于 Docker 和 python 开发，可用于构建分布式边缘计算、网络环境仿真等场景。

功能概览

- 无线网络配置
- 计算资源管理
- 容器动态配置
- 容器计算和网络应用快速构建

开发人员

- 林彬
- 毕宿志

系统环境

ERM V1.0 推荐使用环境（服务器端）

- 软件环境
  - 操作系统：Ubuntu 18.04 aarch64
  - Python 3.6
  - TensorFlow 1.12.0
- 硬件环境
  - CPU：ARM Cortex-A72 1.5GHz（四核）
  - RAM：4GB DDR4
  - 磁盘空间：8GB
  - 无线网卡
- 网络配置

- IP 地址：192.168.191.4
- 接收端口：8080

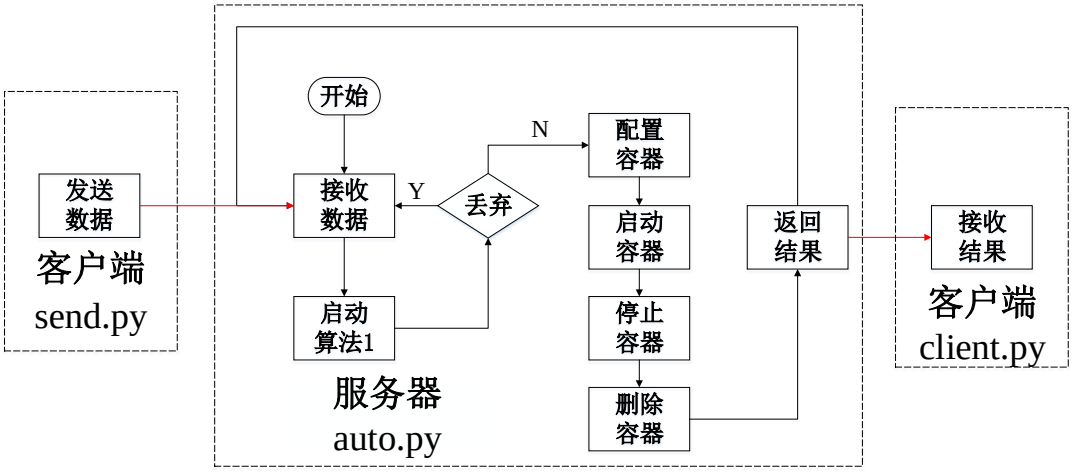
ERM V1.0 推荐使用环境（客户端）

- 软件环境
  - Python 3.6
- 硬件环境
  - 无线网卡
- 网络配置
  - IP 地址：192.168.191.5—192.168.191.8（四个用户依次递推）
  - 接收指令端口（用于启动实验）：8081—8084（四个用户依次递推）
  - 接收结果端口：8085

值得注意的是：在实验中，应根据实际通信的 IP 地址更改代码中的 ip 变量。

整体流程

服务器端只需启动 auto.py 程序，等待用户请求即可。客服端需要打开两个 python 程序：第一个是 send.py，用于发送任务请求；第二个是 client.py，用于接收任务输出结果。具体流程如下：



一些补充：对于客户端发送的数据，其首先需要输入$t_{r,k}$，这个数值将用于指定计算的时延要求（由于构建的传输模型是基于局域网下的无线传播模型（传输速率普遍在 1Mbits 以上），对于单张图片来说（在 1-100Kbits 之间），传输时延是可以忽略不计的）。接下来，用户将指定一个文件路径或者文件夹路径用于上传图片，具体如下：对于文件路径，send.py 程序将判断该路径下的文件是否为

图片，如果是就将其上传到服务器进行识别，不是则告知用户该文件不属于图片，并且不进行传输；对于文件夹路径，send.py 程序将会把文件夹下的所有图片上传到服务器中。此时对于该路径下的所有图片来说，$t_{r,k}$都是相同的。

对于客户端的接收结果，其表现为与图片同名的 txt 文件，并且图像识别的结果就记录在该 txt 文件里面。如果不额外指定，返回结果将和 client.py 居于同一个文件夹下。

对于服务器端，其将基于接收到的比特流长度以及解码后是否为数字型变量，判段接收到的数据是指定的时延要求还是待处理的图片。对于指定的时延要求，服务器会保存下来并作为 CPU 调配算法（见下一节）的输入；对于待处理的图片，服务器会基于数据量近似估计工作负载并用于预测完成时间。

关于容器：容器的生成需要用到镜像，并且一个镜像可以同时生成多个容器，基于这些容器，服务器就可以进行 CPU 的调配（容器也支持运行内存的分割，但由于在本项目中充当服务器的树莓派的底层架构不支持，因此本项目只考虑 CPU 的调配）。

关于镜像：一个完整的镜像应包括运行程序所需要的所有依赖环境以及运行程序本身。对于运行程序（**它可以是任何适用于依赖环境的程序**），在本项目中为用户提供的是图像识别功能（这里用到的图像识别程序是 akshaypai 在 GitHub 开源的 classifier.py）。对于依赖环境，则分成了以下四类：

（1）操作系统：Ubuntu_18.04；

（2）程序运行的平台：python_3.6；

（3）程序运行的依赖包：tensorflow_1.12.0；

（4）容器获取待处理图片的组件：openssh-server。

关于镜像的创建，本项目采用了 Dockerfile 的方式，Dockerfile 内容如下表所示：

Dockerfile
FROM pcarstens8698/tensorflow-1.12.0-py3-none-linux_aarch64:latest MAINTAINER wenting LABEL version="1.0"

```
WORKDIR /classifier_api
ADD . /classifier_api

RUN sed -i 's/ports.ubuntu.com/mirrors.ustc.edu.cn/g' /etc/apt/sources.list \
&& rm -rf /var/lib/apt/lists/* \
&& apt-get update \
&& apt-get install build-essential \
&& apt-get install -y openssh-server
```

之后将 Dockerfile 文件和 classifier.py 置于同一个文件夹下，并在这个文件夹下运行“sudo docker build -t 镜像名 .”即可。（其中镜像名可以任意指定）

CPU 调配算法介绍

部分变量名及其含义			
w_k	第k个任务的工作负载，由服务器根据任务的数据量估计得到	g	松弛因子，g值越大表明计算的节余时间越充足
f	单个 CPU 的频率，在本项目中为 1500MHz	$t_{a,k}$	第k个任务的到达时间
n	CPU 个数，在本项目中为 4	$t_{q,k}$	第k个任务的排队时间
$f_{r,i}$	第i个 CPU 剩余的计算资源	$t_{s,k}$	第k个任务的开始时间
c_k	第k个任务分配的 CPU 频率	$t_{d,k}$	第k个任务的截止时间
i_k	第k个任务分配的 CPU 索引	$t_{f,k}$	第k个任务的预计完成时间
o	锁定了当前 CPU 计算资源的任务索引集合	$t_{r,k}$	第k个任务的时延要求，由用户给定
index	用于对 CPU 的剩余计算资源进行索引排序（从大到小）		

算法 1：CPU 调配算法

Initialize： f，n，g，$k = 0$；

While receiving a request of user：

$k = k + 1$；

Input： w_k，$t_{a,k}$，$t_{q,k}$，$t_{r,k}$；

Initialize： $t_{s,k} = t_{a,k} + t_{q,k}$，$t_{d,k} = t_{a,k} + t_{r,k}$，$i = 0$，$o = \emptyset$；

Repeat：

If $t_{f,k} \neq \emptyset$：

Repeat：

If $t_{f,j} > t_{s,k}$ and $i_j == i$：

$o = o \cup j$；

$j = j + 1$

until $j < k$

$f_{r,i} = f - \sum_{j \in o} c_j$， $i = i + 1$；

until $i \geq n$；

index = argsort$\left(f_{r,i} | \, i < n, i \in N\right)$， index = index[::-1]；

对 CPU 的剩余计算资源进行索引排序（从大到小），以优先对计算资源充足的 CPU 进行利用。

Initialize： $i = 0$；

Repeat：

$t_{s,k} = t_{a,k} + t_{q,k}$， $o = \emptyset$；

If $g\frac{w_k}{t_{r,k}} \leq f_{r,\text{index}[i]}$：

$c_k = g\frac{w_k}{t_{r,k}}$， $t_{f,k} = t_{s,k} + \frac{w_k}{c_k}$， $i_k = \text{index}[i]$；

else：（进入等待机制）

If $t_{f,k} \neq \emptyset$：

Repeat：

If $t_{f,j} > t_{s,k}$ and $i_j == \text{index}[i]$：

$o = o \cup j$；

$j = j + 1$

until $j < k$

If $o \neq \emptyset$：

$o = \text{argsort}\left(t_{f,j} | \, j \in o\right)$；

for j in o：

If $t_{f,j} \geq t_{d,k}$：

break；

$t_{s,k} = t_{f,j}$， $f_{r,\text{index}[i]} = f_{r,\text{index}[i]} + c_j$；

If $g\frac{w_k}{t_{d,k} - t_{s,k}} \leq f_{r,\text{index}[i]}$：

$c_k = g\frac{w_k}{t_{d,k}-t_{s,k}}$，$t_{f,k} = t_{s,k} + \frac{w_k}{c_k}$，$i_k = \text{index}[i]$，**break**；

If $i_k \neq \emptyset$：

break；

If $i == (n-1)$：

$t_{s,k} = +\infty$，$c_k = 0$，$t_{f,k} = 0$，$i_k = -1$；

$i = i + 1$；

until $i \geq n$；

Output：c_k，i_k。

特性：①考虑到多核 CPU 与单核 CPU 对计算时间的不同影响，这里为了保证 CPU 频率与计算时间的反比关系（这种反比关系在单核 CPU 的调配下可以实现。例如，在分配了 900MHz 的单核 CPU 下，识别一张图片需要大约 34 秒；而对于 1500MHz 的满载单核 CPU 来说，其需要 22 秒），将不对多核 CPU 计算同个任务的情况加以考虑。这是因为，在实际中，如果基于 1500MHz 的单核 CPU 对一张图片进行识别，其计算时间大概是 22 秒（在本项目所用的树莓派环境下）；然而在 1500MHz 的双核 CPU 环境下，其计算时间仅能达到 18 秒，并不能实现计算时间的同比下降。

②引入了等待机制，在当前 CPU 不足以处理任务请求时，服务器（即树莓派 0）会基于预测的完成时间进行等待判定：这部分判定具体见算法 1；并且这部分资源会被锁定，不会因为其他任务的到来而分配给其他任务使用。

③只要接收到用户的任务请求，服务器就会基于算法 1 进行决策（具体可分成三类：不等待直接分配 CPU、等待后再分配 CPU 以及丢弃当前任务请求），并根据决策结果配置并启动容器（该容器集成了运行程序所需要的环境，其是由镜像生成的）。

性能评估

为了评估 CPU 调配算法的性能，这里提出两种基准方案以供比较。

第一种为贪婪算法（greedy algorithm），其核心思想是对于到来的每一张图片，都尽全力计算。当需要处理两张图片时，容器将以相同的权重竞争 CPU 资源（即平分计算资源）。对于同时处理三张图片，计算资源则三等分，以此类推。在这个算法中，由于涉及到多核 CPU 共同处理相同任务的情况，因此不对完成时间

进行预测。

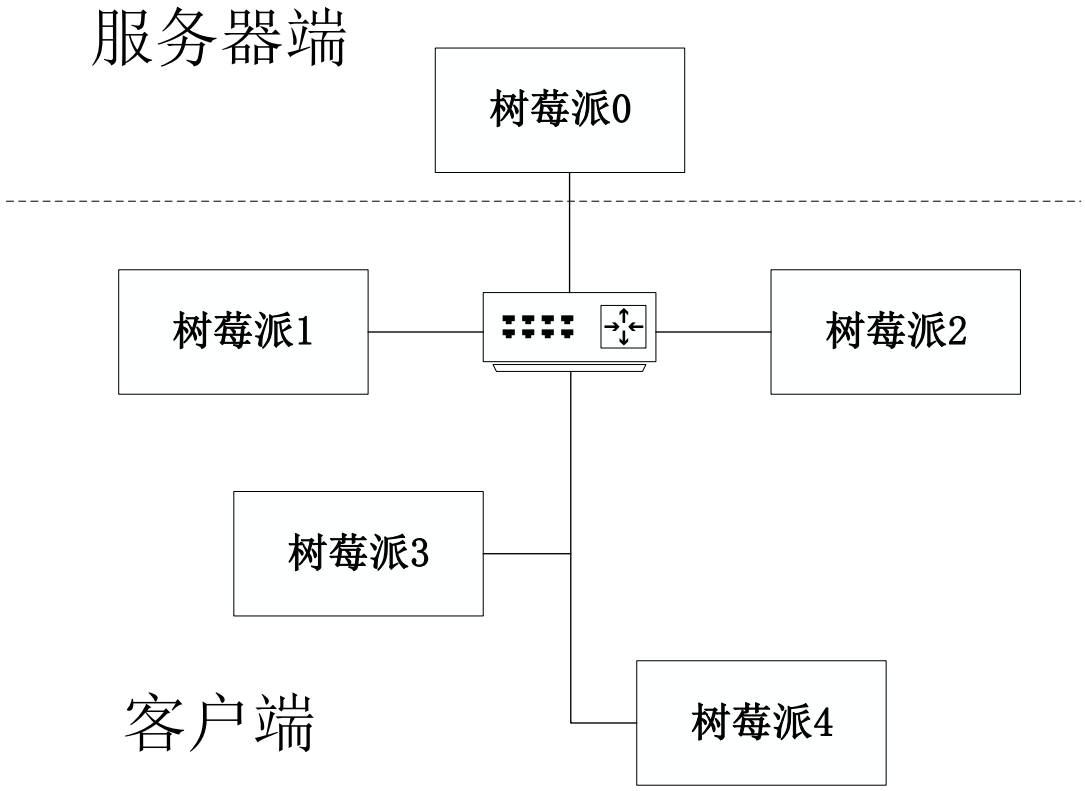
第二种为固定资源算法（fixed algorithm），其核心思想是对于到来的每一张图片，都以一个固定的 CPU 频率（在本项目中表现为 750MHz，也就是说服务器可以并发地处理 8 张图片）进行计算。

CPU 调配的目的是尽可能多地满足任务的时延要求，基于此我们提出了以下性能评估指标——超时任务数，具体定义如下：记录实际的计算时间，将其与任务的时延要求比较。如果超过时延要求，则将此任务定义为超时，并记录，最后对超时任务进行统计，得到总的超时任务数。

在本项目的性能评估中，所用到的系统模型如下图所示。为了体现多用户场景下任务需求的时变特性，在本项目中用到了五个树莓派一同完成性能评估实验。具体地，我们将树莓派 0 作为服务器，用于提供图像识别功能；对于客户端，则由 4 个树莓派组成。

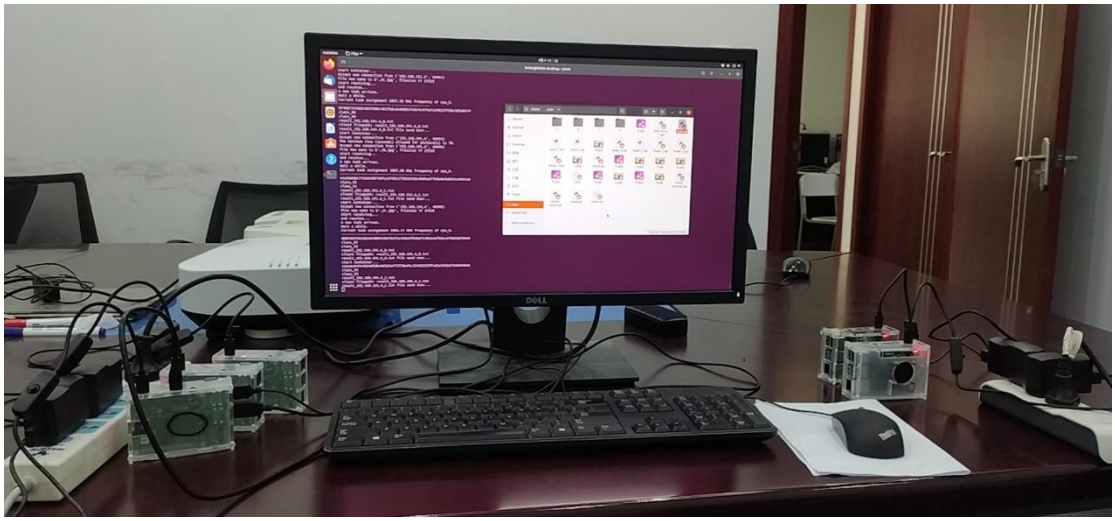
值得注意的一点是：在本实验中，虽然用于图像识别的图片均为同一张图片，但是可以通过改变同一时刻发送的图片数量来实现工作负载的差异性。具体地，对于树莓派 1，每个时间间隔仅发送一张图片，时延要求假定为 50 秒；对于树莓派 2，发送两张图片，时延要求假定为 70 秒；对于树莓派 3，发送三张图片，时延要求假定为 200 秒；对于树莓派 4，发送 2~7 张图片，时延要求假定为 500 秒。此外，对于这些树莓派的每一次发送间隔，依次设置为 50 秒、70 秒、200 秒、500 秒。实验时长设定为 20 分钟，在此期间内客服端将依据设定的发送间隔进行图片传输。

需要用到的实验设备：一根网线、一台充当无线路由器的笔记本电脑、五个树莓派（通过 WIFI 信号连接到同一个局域网下）、一个显示器、5 个二孔插座（用于树莓派供电）、2 个三孔插座（用于显示器以及笔记本供电）、一个键盘、一个鼠标。

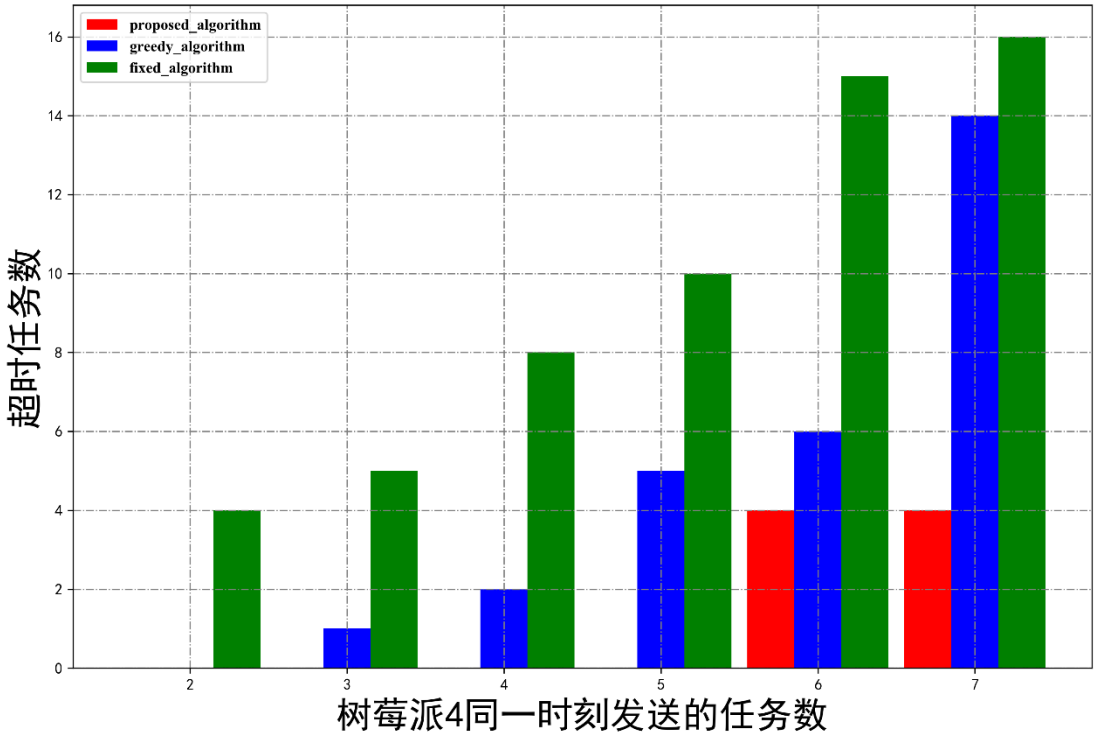


实验步骤：以树莓派 1 为例，首先将 user1's_code_and_picture 文件夹下的图片文件夹及代码置于树莓派 1，利用 python3.6 启动 begin_1.py 等待服务器端发送“start”命令，并在另一个终端打开 client.py 用于接收图像识别结果。对于其他充当客服端的树莓派 2、3、4，实验操作类似。

对于服务器端，其中 auto.py 代表了 CPU 调配算法下的服务器，auto_1.py 代表了贪婪算法下的服务器，auto_2.py 代表了固定资源算法下的服务器。首先将 server_code 文件夹下的代码置于树莓派 0，以 CPU 调配算法下的服务器为例，利用 python3.6 启动 auto.py 用于接收任务请求——分配计算资源——启动容器——返回计算结果，并在另一个终端打开 begin_0.py 告知客户端实验开始，可以进行图片传输。实际的实验架构如下图所示：



实验结果如下图所示：



对于树莓派 4 同一时刻发送的任务数可以通过对 user4's_code_and_picture/4 文件夹下的图片数目进行删减或增添来实现同时传输任务数的减小以及增大。