

分布式作业

第二次作业

姓名：谢宇桐

班级：计科教学二班

学号：22336259

一、理论

- 我们以书上的例子read，上网查询过程后进行说明：

```
count = read(fd, buf, nbytes)
```

其中fd为一个整型数表示一个文件，buf为一个字符数组，用于存储读入的数据，nbytes为另一个整型数，用来记录实际读入的字节数。

首先**客户过程以正常的方式调用客户存根**，请求从文件中读取数据；

```
// 客户端代码
// ... 先对上面三个变量进行赋值 ...

// 调用客户存根
int result = read(fd, buf, &nbytes);
```

客户存根生成一个消息,然后调用本地操作系统;

```
// 客户存根代码
int read(int fd, char* buf, int* nbytes) {
    // 创建消息, 包含文件描述符、缓冲区指针和字节数指针
    RPC_Message message;
    message.fd = fd;
    message.buf = buf;
    message.nbytes_ptr = nbytes;

    // 发送消息
    return send_rpc_message("read", &message);
}
```

客户端操作系统通过网络将消息发送给远程操作系统;

远程操作系统将消息交给服务器存根;

服务器存根将参数提取出来,然后调用服务器; 服务器存根解析消息，提取参数，并调用服务器程序。

```
// 服务器存根代码
void handle_rpc_message(RPC_Message* message) {
    // 提取参数
    int fd = message->fd;
    char* buf = message->buf;
    int* nbytes = message->nbytes_ptr;

    // 调用服务器程序
    int result = server_read(fd, buf, nbytes);

    // ...将结果打包成消息并发送结果消息...
}
```

服务器执行要求的操作,操作完成后将结果返回给服务器存根;

```
// 服务器代码
int server_read(int fd, char* buf, int* nbytes) {
    // 根据文件描述符读取数据到缓冲区
    int result = actual_read(fd, buf, *nbytes);
    *nbytes = result; // 更新实际读取的字节数
    return result;
}
```

服务器存根将结果打包成一个消息,然后调用本地操作系统;

服务器操作系统将含有结果的消息发送回客户端操作系统;

客户端操作系统将消息交给客户存根;

客户存根将结果从消息中提取出来,返回给调用它的客户过程。

```
// 客户存根代码
int handle_rpc_result(RPC_Result* result) {
    return result->result;
}
```

```
// 客户端代码
int main() {
    // ...三个变量赋值...
    int result = read_rpc(fd, buf, &nbytes);
    // 使用读取的数据
    // ...
    return 0;
}
```

2. 在客户端和服务端之间，数据被分割成小块（如果需要的话），然后通过套接字发送。接收方的套接字会接收这些数据块，并将它们重新组合成原始数据。

服务器端（一般按顺序执行前四个）：

创建新的通信端点（Socket）：

服务器端首先需要创建一个套接字。这通常通过调用 `socket()` 函数完成，该函数会返回一个套接字描述符，用于后续的所有网络操作。

将本地地址附加到套接字上 (Bind) :

服务器需要将套接字绑定到一个特定的IP地址和端口上，以便客户端可以找到它。

宣布已准备好接受连接 (Listen) :

服务器调用 `listen()` 函数，让套接字进入监听状态，准备接受客户端的连接请求。

在收到连接请求之前阻塞调用方 (Accept) :

当客户端发起连接请求时，服务器通过 `accept()` 函数接受连接，这会创建一个新的套接字专门用于与该客户端通信。

数据通信 (Send/Receive) :

一旦连接建立，服务器就可以使用 `receive()` 函数来读取客户端发送的数据，使用 `send()` 函数来向客户端发送数据。

释放连接 (Close) :

数据传输完成后，服务器可以调用 `close()` 函数关闭与客户端的连接。

-

客户端:

创建新的通信端点 (Socket) :

客户端也需要创建一个套接字，通过 `socket()` 函数完成。

主动尝试确立连接 (Connect) :

客户端使用 `connect()` 函数发起对服务器的连接请求，需要提供服务器的IP地址和端口号。

数据通信 (Read/Write) :

连接成功后，客户端可以使用 `read()` 函数来接受数据，使用 `write()` 来向服务器发送数据。

关闭连接 (Close) :

通信完成后，客户端也需要调用 `close()` 函数来关闭连接。

-

-

3. 主要缺点是它要求每台名称服务器都具有较高的性能。还有以下缺点：可能会进行重复计算，这会浪费时间和资源，降低查询效率；内存消耗过大；可能会导致网络资源的大量消耗；可能导致查询的延迟增加。

-

-

4. 通常是状态相关的，理由如下：

- 连接管理：TCP是一种面向连接的协议，在数据传输发生之前，必须在客户端和服务端之间建立一个连接。服务器需要维护每个连接的状态信息，以确保数据正确地发送和接收。
- 数据顺序：由于TCP保证数据包的顺序，服务器需要跟踪每个连接的数据包顺序，以确保数据可以按正确的顺序重新组装。

- 流量控制：TCP使用窗口机制进行流量控制，服务器需要为每个连接维护窗口大小的状态，以控制数据的发送速率。
- 连接终止：服务器需要能够识别并正确响应连接终止请求，这涉及到维护连接的状态信息，直到连接被完全关闭。
- 错误恢复：如果发生错误，如丢包或乱序，服务器需要能够恢复，这通常涉及到重传机制，需要服务器知道连接的状态以进行适当的恢复操作。

二、实验

(一)

1. 问题描述

CRIU是一种在用户空间实现的进程或者容器checkpoint和restore的方法，从而实现进程或者容器的保存和恢复。请利用CRIU实现进程和容器的迁移（迁移种类不限），并测试迁移过程中的性能损耗（如进程停止时间、网络传输时间等）。

2. 解决方案与实验结果

(1) 环境配置

<1> docker

```
# 卸载旧版本
sudo apt-get remove docker docker-engine docker.io
# 更新apt包索引
sudo apt-get update
# 安装nfs
sudo apt-get install nfs-kernel-server
# 安装docker相关支持文件
sudo apt-get install docker-ce docker-ce-cli containerd.io
sudo apt install build-essential -f -y
sudo apt install pkg-config -f -y
sudo apt install libnet-dev python-yaml libaio-dev -f -y
sudo apt install libprotobuf-dev libprotobuf-c-dev protobuf-c-compiler protobuf-
compiler python-protobuf libnl-3-dev libcap-dev libbsd-dev -f -y
sudo apt-get install \
    apt-transport-https \
    ca-certificates \
    curl \
    gnupg-agent \
    software-properties-common -y -f
# 安装docker
# 用下面这句添加密钥时遇到问题①，将在后面进行说明。
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
# add repo
sudo add-apt-repository \
    "deb [arch=amd64] https://download.docker.com/linux/ubuntu \
    $(lsb_release -cs) \
    stable"
# update and install
sudo apt-get update
```

```
sudo apt-get install docker-ce docker-ce-cli containerd.io  
# 添加镜像源和开启experimental属性  
# 使用nano编辑器进行编辑  
sudo nano /etc/docker/daemon.json
```

添加如下配置：

```
{  
  "registry-mirrors": ["https://docker.sysumsc.cn"],  
  "experimental": true  
}
```

按 Ctrl + X, 然后按 Y 确认保存更改, 最后按 Enter 键退出。

```
# 重启docker  
sudo systemctl restart docker  
  
#检查是否安装完成  
docker -v  
sudo docker run hello-world
```

终于安装完成docker

```
xyt@ubuntu:~$ docker -v  
Docker version 24.0.2, build cb74dfc
```

```
xyt@ubuntu:~$ sudo docker run hello-world
```

```
Hello from Docker!  
This message shows that your installation appears to be working correctly.  
  
To generate this message, Docker took the following steps:  
1. The Docker client contacted the Docker daemon.  
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.  
   (amd64)  
3. The Docker daemon created a new container from that image which runs the  
   executable that produces the output you are currently reading.  
4. The Docker daemon streamed that output to the Docker client, which sent it  
   to your terminal.  
  
To try something more ambitious, you can run an Ubuntu container with:  
$ docker run -it ubuntu bash  
  
Share images, automate workflows, and more with a free Docker ID:  
https://hub.docker.com/  
  
For more examples and ideas, visit:  
https://docs.docker.com/get-started/
```

-

<2> CRIU

```
# criu install
curl -O -SSL http://download.openvz.org/criu/criu-3.14.tar.bz2
tar xjf criu-3.14.tar.bz2
cd criu-3.14
make
sudo cp ./criu/criu /usr/local/bin
# check
sudo criu check
```

最后显示looks good 则说明安装没问题

```
xyt@ubuntu:~/criu-3.14$ sudo criu check
Looks good.
```

(2) 进程迁移

编写一个每秒输出一个递增的数字的简单 C 语言程序实现一个循环。

```
gedit main.c # 打开main.c进行编辑
```

```
// main.c
#include <stdio.h>
#include <unistd.h>

int main(int argc, char const *argv[]){
    int i = 0;
    for(; i < 1000; ++i){
        sleep(1);
        printf("%d\n", i);
    }
    return 0;
}
```

```
# 编译为myprogram文件
gcc -o myprogram main.c
# 运行
./myprogram
```

```
xyt@ubuntu:~$ gedit main.c
xyt@ubuntu:~$ gcc -o myprogram main.c
xyt@ubuntu:~$ ./myprogram
0
1
2
3
4
```

```
xyt@ubuntu:~$ gcc -o myprogram main.c
xyt@ubuntu:~$ ./myprogram
0
1
2
3
4
5
6
7
8
9
10
11
12
```

保持这个程序运行的同时打开另一个新终端，查找示例程序的进程号，通过 criu 设置应用的 checkpoint:

```
xyt@ubuntu: ~  
File Edit View Search Terminal Help  
File Edit View xyt@ubuntu:~$ ps -ef | grep myprogram  
12 xyt      2724    2715  0 21:32 pts/1    00:00:00 grep --color=auto myprogram  
13 xyt@ubuntu:~$ sudo criu dump -D/home/xyt -j -t 2724  
14 [sudo] password for xyt:  
15 Warn (compel/src/lib/infect.c:127): Unable to interrupt task: 2724 (No such p  
16 rocess)  
17 Error (compel/src/lib/infect.c:346): Unable to detach from 2724: No such proce  
18 ss  
19 Error (criu/cr-dump.c:1764): Dumping FAILED.  
20 xyt@ubuntu:~$ ps -ef | grep myprogram  
21 xyt      2807    2673  0 21:34 pts/0    00:00:00 ./myprogram 1.查找进程号  
22 xyt      2809    2715  0 21:34 pts/1    00:00:00 grep --color=auto myprogram  
23 xyt@ubuntu:~$ sudo criu dump -D/home/xyt -j -t 2807  
24 Warn (compel/arch/x86/src/lib/infect.c:281): Will restore 2807 with interrupt  
25 ed system call  
26 xyt@ubuntu:~$ 2.设置checkpoint  
27  
28  
29  
30  
31  
32  
33  
34 Killed 3.进程被杀死  
xyt@ubuntu:~$
```

ps -ef：用来列出系统中所有运行中的进程的命令。

- ps 是“Process Status”的缩写，用于显示当前运行的进程状态。
- -e 选项表示显示所有进程。
- -f 选项表示全格式输出，包括更详细的信息，如用户ID、进程ID、父进程ID、进程名等。

-

- criu dump：用于创建进程快照。
- -D：指定快照文件的存储目录。
- -j：表示该进程是通过shell启动的作业。在CRIU中，如果进程是通过shell启动的，需要加上这个选项以便正确地处理作业控制信号。
- -t：指定要快照的进程ID (PID)。

-

我们再将它还原

注意：checkpoint 时指定的应用程序是由 shell 启动，所以 restore 时需要指定相应的 -j 选项。


```
xyt@ubuntu: ~  
File Edit View Search Terminal Help  
20  
21  
22  
23  
24  
25  
26  
27  
28  
29  
30  
31  
32  
33  
34  
Killed  
xyt@ubuntu:~$ sudo criu restore -D /home/xyt -j  
[sudo] password for xyt:  
35  
36  
37  
38  
39  
40
```

可以发现程序从上次被杀死的时间点继续运行，而此时看它的进程号仍使用原来的进程号

```
xyt@ubuntu:~$ ps -ef | grep myprogram  
xyt      2807    2673  0 21:34 pts/0    00:00:00 ./myprogram  
xyt      2809    2715  0 21:34 pts/1    00:00:00 grep --color=auto myprogram  
xyt@ubuntu:~$ sudo criu dump -D/home/xyt -j -t 2807  
Warn (compel/arch/x86/src/lib/infect.c:281): Will restore 2807 with interrupt  
ed system call  
xyt@ubuntu:~$ ps -ef | grep myprogram  
xyt      2807    2830  0 21:36 pts/0    00:00:00 ./myprogram  
xyt      2810    2715  0 21:37 pts/1    00:00:00 grep --color=auto myprogram  
xyt@ubuntu:~$
```

性能损耗——时间:

先让程序跑起来并获取myprogram的pid后（6674），我们写一个sh自动脚本loss_test.sh测试时间:

```
#!/bin/bash

PID=$(pgrep myprogram)

# 创建checkpoint
sudo criu dump -D /home/xyt -j -t $PID --display-stats

# 恢复checkpoint
sudo criu restore -D /home/xyt -j --display-stats
```

```
9
10
Killed
xyt@ubuntu:~$ ./myprogram
0
1
2
3
4
5
6
7
Killed
xyt@ubuntu:~$
xyt@ubuntu:~$ gedit loss_test.sh
xyt@ubuntu:~$ ./loss_test.sh
Warn (compel/arch/x86/src/lib/infect.c:281): Will restore 7043 with interrupted system call
Displaying dump stats:
Freezing time: 407 us
Frozen time: 73339 us
Memory dump time: 13093 us
Memory write time: 644 us
IRMAP resolve time: 0 us
Memory pages scanned: 1129 (0x469)
Memory pages skipped from parent: 0 (0x0)
Memory pages written: 23 (0x17)
Lazy memory pages: 0 (0x0)
Displaying restore stats:
Pages compared: 0 (0x0)
Pages skipped COW: 0 (0x0)
Pages restored: 23 (0x17)
Restore time: 12297 us
Forking time: 1 us
8
9
10
11
```

这些指标中冻结时间（Freezing time）和内存转储时间（Memory dump time）这两个值直接反映了CRIU操作的快慢，我们可以看到分别是407us和13093us。内存写入时间（Memory write time）则反映了存储系统的性能，对于性能损耗也有很大影响。总的来说，这些时间越短，表示CRIU操作的性能损耗越小，checkpoint操作的效率越高。

-

-

(3) 容器迁移

先创建容器，名为looper2

```
sudo docker run -d --name looper2 --security-opt seccomp:unconfined busybox
/bin/sh -c 'i=0; while true; do echo $i; i=$((expr $i + 1)); sleep 1; done'
```

- `docker run` 用于创建和运行一个新的 docker 容器looper2。
- `-d` 表示Docker在“分离模式”下运行容器，即容器在后台运行。
- `--security-opt seccomp:unconfined`：这是一个安全选项，用于设置容器的安全配置。
seccomp (secure computing mode) 是一种Linux内核功能，用于限制进程可执行的系统调用。
unconfined参数表示不对容器进行seccomp限制，即允许容器内的进程执行所有系统调用。
- `busybox`：要使用的Docker镜像的名称。
- `/bin/sh -c`：容器启动后执行的命令。`/bin/sh` 是BusyBox提供的shell，`-c` 参数表示后面跟着的是要执行的命令字符串。
- `'i=0; while true; do echo $i; i=$((expr $i + 1)); sleep 1; done'` 是在容器上要执行的命令，使用无限循环，每个循环输出一个递增的数字，等待 1 秒钟然后继续即 命令会不断输出数字，直到手动停止容器。

-

添加checkpoint

```
sudo docker checkpoint create looper2 checkpoint1
```

不加sudo会显示permission denied, 若无报错不加sudo也可

```
xyt@ubuntu:~$ docker checkpoint create looper2 checkpoint1
permission denied while trying to connect to the Docker daemon socket at unix:///var/run/docker.sock: Post "http://%2Fvar%2Frun%2Fdocker.sock/v1.24/containers/looper2/checkpoints": dial unix /var/run/docker.sock: connect: permission denied
xyt@ubuntu:~$ sudo docker checkpoint create looper2 checkpoint1
checkpoint1
```

查看容器状态:

```
/var/run/docker.sock: connect: permission denied
xyt@ubuntu:~$ sudo docker ps -a
```

| CONTAINER ID | IMAGE | COMMAND | CREATED | STATUS |
|--------------|-------------|--------------------------|----------------|----------------------------|
| d7409db741c2 | busybox | "/bin/sh -c 'i=0; wh..." | 16 minutes ago | Exited (137) 2 minutes ago |
| 51b236ed9543 | hello-world | "/hello" | 13 hours ago | Exited (0) 13 hours ago |
| 5ae6e738b31c | hello-world | "/hello" | 13 hours ago | Exited (0) 13 hours ago |

```
xyt@ubuntu:~$
```

```
File Edit View Search Terminal Help
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
xyt@ubuntu:~$
```

可以看到容器状态变为exited，且数字循环停在847。

接下来我们将其还原：

```
sudo docker start --checkpoint checkpoint1 looper2
```

我们看到它的状态变为up

```
xyt@ubuntu:~$ sudo docker start --checkpoint checkpoint1 looper2
[sudo] password for xyt:
xyt@ubuntu:~$ sudo docker ps -a
```

| CONTAINER ID | IMAGE | COMMAND | CREATED | STATUS |
|--------------|--------------------|--------------------------|--------------|-----------------------|
| d7409db741c2 | busybox looper2 | "/bin/sh -c 'i=0; wh..." | 3 hours ago | Up 19 seconds |
| 516236ed9543 | hello-world | "/hello" | 16 hours ago | Exited (0) 16 hours a |
| go | nice_panini | | | |
| 5ae6e738b31c | hello-world | "/hello" | 16 hours ago | Exited (0) 16 hours a |
| go | charming_pare | | | |

File Edit View Search Terminal Help

1008

1009

1010

1011

1012

1013

1014

1015

1016

1017

1018

1019

1020

1021

1022

1023

1024

1025

1026

1027

1028

1029

1030

xyt@ubuntu:~\$

我们可以看到它从checkpoint继续运行，且不断更新

```
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
xyt@ubuntu:~$
```

性能损耗:

在运行容器时，我们通过另一个终端执行以下代码进行实时监控容器状态:

```
sudo docker stats loop2
```

中断前:

| CONTAINER ID | NAME | CPU % | MEM USAGE / LIMIT | MEM % | NET I/O |
|--------------|-------|-------|--------------------|-------|-------------|
| d7409db741c2 | loop2 | 0.25% | 2.055MiB / 3.81GiB | 0.05% | 3.37kB / 0B |
| BLOCK I/O | PIDS | | | | |
| 0B / 0B | 2 | | | | |

| CONTAINER ID | NAME | CPU % | MEM USAGE / LIMIT | MEM % | NET I/O |
|--------------|-------|-------|--------------------|-------|-------------|
| d7409db741c2 | loop2 | 0.26% | 1.953MiB / 3.81GiB | 0.05% | 3.64kB / 0B |
| BLOCK I/O | PIDS | | | | |
| 0B / 0B | 2 | | | | |

中断时:

| CONTAINER ID | NAME | CPU % | MEM USAGE / LIMIT | MEM % | NET I/O | BLO |
|--------------|-------|-------|-------------------|-------|---------|-----|
| d7409db741c2 | loop2 | 0.00% | 0B / 0B | 0.00% | 0B / 0B | 0B |
| CK I/O | PIDS | | | | | |
| / 0B | 0 | | | | | |

恢复后:

| CONTAINER ID | NAME | CPU % | MEM USAGE / LIMIT | MEM % | NET I/O |
|--------------|---------|-------|-------------------|-------|-------------|
| BLOCK I/O | PIDS | | | | |
| d7409db741c2 | looper2 | 0.19% | 2.59MiB / 3.81GiB | 0.07% | 3.37kB / 0B |
| 0B / 0B | 2 | | | | |

| CONTAINER ID | NAME | CPU % | MEM USAGE / LIMIT | MEM % | NET I/O |
|--------------|---------|-------|--------------------|-------|-------------|
| BLOCK I/O | PIDS | | | | |
| d7409db741c2 | looper2 | 0.51% | 2.594MiB / 3.81GiB | 0.07% | 3.64kB / 0B |
| 0B / 0B | 2 | | | | |

^C

从这部分数据来看:

CPU使用率: 中断前的CPU使用率是0.25%和0.26%, 中断后的CPU使用率是0.51%和0.19%。这表明迁移过程中没有显著的性能损耗。

内存使用情况: 中断前的内存使用量是2.055MiB和1.953MiB, 中断后的内存使用量是2.594MiB和2.59MiB。内存使用量在迁移后有所增加, 这可能表明迁移过程中有额外的内存分配或复制。

网络I/O: 中断前后的网络I/O变化不大, 网络输入/输出流量在迁移前后基本保持一致。

块I/O: 中断前后的块I/O都是0B / 0B, 表明迁移过程中没有显著的磁盘活动。

进程数 (PIDs): 中断前后的进程数都是2, 表明迁移过程中进程数没有变化。

综合这些数据, 我们可以得出结论: 虽然内存使用量有所增加, 但CPU使用率、网络I/O和块I/O没有显著变化, 进程数保持一致。因此, 从这些数据来看, 容器迁移过程中的性能损耗不大。

3. 遇到的问题及解决方法

问题①: 在安装docker时经常遇到网络限制使得有些工具没有安装到, 导致无法安装docker, 如:

```
xyt@ubuntu:~$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
curl: (35) OpenSSL SSL_connect: SSL_ERROR_SYSCALL in connection to download.docker.com:443

gpg: no valid OpenPGP data found.
xyt@ubuntu:~$

xyt@ubuntu:~$ sudo apt-get install docker-ce=18.03.1~ce-0~ubuntu
Reading package lists... Done
Building dependency tree
Reading state information... Done
E: Version '18.03.1~ce-0~ubuntu' for 'docker-ce' was not found
xyt@ubuntu:~$
```

上网查询发现大多都是网络问题, 我尝试更换校园网到个人热点, 再重复多次执行命令直至没有报错, 最后终于将docker安装完成。

问题②: 在测试性能损耗的时间时, 我原本使用的代码脚本如下:

```
#!/bin/bash

PID=$(pgrep myprogram)
```

```

# 记录checkpoint开始前的时间
start_time=$(date +%s)

# 创建checkpoint
sudo criu dump -D /home/xyt -j -t $PID --display-stats

# 记录checkpoint完成的时间
end_time=$(date +%s)
echo "Checkpoint duration: $((end_time - start_time)) seconds"

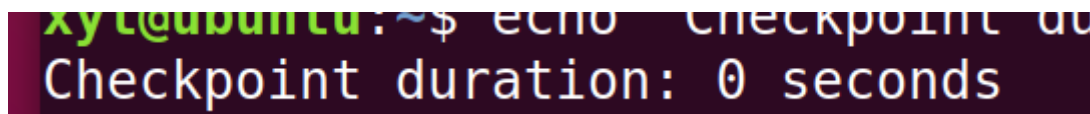
# 记录restore开始前的时间
start_time=$(date +%s)

# 恢复checkpoint
sudo criu restore -D /home/xyt -j --display-stats

# 记录restore完成的时间
end_time=$(date +%s)
echo "Restore duration: $((end_time - start_time)) seconds"

```

但这个只能看损耗的时间，而且单位为秒，损耗一般都在微秒级别，因此这个并不适用。



```

xyt@ubuntu:~$ echo Checkpoint duration: 0 seconds
Checkpoint duration: 0 seconds

```

而在dump和restore命令后面加 `--display-stats`，能详细显示包括dump和restore操作在内的统计信息，如处理的页面数和操作所需的时间。

-

-

(二)

1. 问题描述

使用protobuf和gRPC或者其他RPC框架实现消息订阅（publish-subscribe）系统，该订阅系统能够实现简单的消息传输，还可以控制消息在服务器端存储的时间。

-

2. 解决方案

首先配置环境：

```

pip install grpcio
pip install grpcio-tools googleapis-common-protos

```

首先按照如下文件组织创建rpc文件夹


```
.└─ rpc
.└─└─ pubsub_pb2.py
.└─└─ pubsub_pb2_grpc.py
.└─└─ pubsub.proto
└─ client.py
└─ server.py
```

接下来创建编写pubsub.proto文件

code pubsub.proto

代码如下:

```
syntax = "proto3";

package rpc_package;

// 定义服务
service pubsub {
    // 定义服务的接口
    rpc pubsubServe (mes2server) returns (mes2client) {}
}

// 定义上述接口的参数数据类型
message mes2server {
    string mes1 = 1;
}

message mes2client {
    string mes2 = 1;
}
```

通信时，客户端向服务器发送消息mes2server，服务器返回消息mes2client给客户端。这两个消息在之后生成的代码中会以结构体的形式保存。之后的message定义了mes2server和mes2client结构体的数据，二者都为字符串。

在rpc路径下运行以下指令，使用 `grpc_tools.protoc` 生成工具生成对应语言的库函数，并编译生成 `pubsub_pb2`和`pubsub_pb2_grpc`两个py文件：

```
python -m grpc_tools.protoc -I=./ --python_out=./ --grpc_python_out=./
./pubsub.proto
```

各个参数的功能如下：

- `-m grpc_tools.protoc`表示使用`grpc_tools.protoc`的库模块
- `-I=./`设定源路径为当前文件夹下
- `--python_out=./`表示输出的pb2模块为py文件，输出位置为当前文件夹
- `--grpc_python_out=./`表示输出的pb2_grpc模块为py文件，输出位置为当前文件夹
- `./pubsub.proto`指出了proto文件所在的路径

接下来编写服务器server代码：

```
import threading
from concurrent import futures
import grpc
# 从之前生成的模块中导入消息定义和相关的函数：
from rpc.pubsub_pb2_grpc import add_pubsubServicer_to_server, pubsubServicer
from rpc.pubsub_pb2 import mes2client, mes2server
import time

# 定义服务器类
class PubsubServer(pubsubServicer):
    def __init__(self):
        self.threadLock = threading.Lock() # 其中threadLock为实现线程互斥机制的锁，之后通过锁来控制对消息的输入和线程的阻塞等待输入。n为标志位，起到类似信号量的作用，表示消息是否能输入。n==1表示消息已经被输入，否则n==0，消息还没有输入，需要一个线程发起消息输入的命令，其他所有线程阻塞等待。等n==1时，输入完成，所有线程将输入的消息发给对应的客户端并且将n改为0。mes为需要发送的消息。
        self.n = 0
        self.mes = "default"
        self.mes_timestamp = 0 # 记录消息的时间戳
        self.ttl = 10 # 设置消息的TTL为10秒，指定消息的存活时间

    # 实现proto文件中定义的服务方法
    def pubsubServe(self, request, context):
        # 如果没有消息，则等待消息输入。等待期间，线程锁不会被释放
        if self.n == 0:
            with self.threadLock:
                self.n += 1
                self.mes = input('mes:')
                self.mes_timestamp = time.time() # 更新消息的时间戳
            with self.threadLock:
                current_time = time.time()
                if current_time - self.mes_timestamp > self.ttl: # 查当前时间与消息接收时间间的差值是否超过了 ttl。
                    self.mes = "Message has expired" # 如果超过了 ttl，我们将消息更新为 "Message has expired"。这样，服务器就会在消息超过指定的存活时间后自动将其标记为过期，并发送过期消息给客户端
                    self.mes_timestamp = current_time
                self.n = 0
            return mes2client(mes2=self.mes)

# 创建服务器实例
def serve():
    server = grpc.server(futures.ThreadPoolExecutor(max_workers=3)) # ThreadPoolExecutor创建线程池，max_workers=3表示只有3个线程，也就是最多有3个客户端与服务器相连。
    # 将服务添加到服务器
    add_pubsubServicer_to_server(PubsubServer(), server)
    # 设置服务器监听的端口
    server.add_insecure_port('[::]:50000')
    # 启动服务器
    server.start()
    server.wait_for_termination()
```

```
if __name__ == '__main__':  
    serve()
```

编写客户端client代码：

```
import grpc  
from rpc.pubsub_pb2 import mes2client, mes2server  
from rpc.pubsub_pb2_grpc import pubsubStub  
  
def run():  
    # 配置通信的服务器IP地址和端口  
    with grpc.insecure_channel('localhost:50000') as channel:  
        # 创建客户端存根，用于调用服务器上的方法。  
        stub = pubsubStub(channel)  
  
        while True: # 添加无限循环，持续接收消息  
            # 向服务器发送请求，等待服务器的响应  
            response = stub.pubsubServe(mes2server(mes1='client'), timeout=500)  
  
            # 打印服务器返回的消息  
            print("Server response: " + response.mes2)  
  
if __name__ == '__main__':  
    run()
```

3. 实验结果

先运行服务器程序：

```
python server.py
```

```
D:\分布式\22336259-谢宇桐-作业2\experiment2>python server.py  
C:\Users\85013\anaconda3\Lib\site-packages\google\protobuf\runtime_version.py:112: UserWarning: Protobuf gencode version  
5.27.2 is older than the runtime version 5.28.2 at pubsub.proto. Please avoid checked-in Protobuf gencode that can be o  
bsolete.  
  warnings.warn(
```

(上述warning只是工具版本有些问题，不影响本次实验结果)

因为目前没有任何客户端连接，服务器不会有任何反应。

然后开启客户端：

```
python client.py
```

因为最多开启三个客户端，我全部开启了。当有客户端开启时，服务器就会产生输入消息的请求


```
C:\WINDOWS\system32\cmd x + v
KeyboardInterrupt
Exception ignored in: <module 'threading' from 'C:\Users\85013\anaconda3\Lib\threading.py'>
Traceback (most recent call last):
  File "C:\Users\85013\anaconda3\Lib\threading.py", line 1592, in _shutdown
    _atexit._atexit()
  File "C:\Users\85013\anaconda3\Lib\concurrent\futures\thread.py", line 31, in _python
    _exit()
  File "C:\Users\85013\anaconda3\Lib\threading.py", line 1147, in join
    self._wait_for_tstate_lock()
  File "C:\Users\85013\anaconda3\Lib\threading.py", line 1167, in _wait_for_tstate_lock
    if lock.acquire(block, timeout):
KeyboardInterrupt
KeyboardInterrupt
C
D:\分布式\experiment2>python server.py
C:\Users\85013\anaconda3\Lib\site-packages\google\protobuf\runtime_version.py:112: User
Warning: Protobuf gencode version 5.27.2 is older than the runtime version 5.28.2 at pu
bsub.proto. Please avoid checked-in Protobuf gencode that can be obsolete.
warnings.warn(
mes:it's a message that only two clients can receive
mes:

C:\WINDOWS\system32\cmd x + v
locking
event = call.next_event()
File "src\python\grpcio\grpc\cython\cygrpc\channel.pyx.pxi", line 388, in grpc
_cython.cygrpc.SegregatedCall.next_event
File "src\python\grpcio\grpc\cython\cygrpc\channel.pyx.pxi", line 211, in grpc
_cython.cygrpc._next_call_event
File "src\python\grpcio\grpc\cython\cygrpc\channel.pyx.pxi", line 205, in grpc
_cython.cygrpc._next_call_event
File "src\python\grpcio\grpc\cython\cygrpc\completion_queue.pyx.pxi", line 78,
in grpc._cython.cygrpc._latent_event
File "src\python\grpcio\grpc\cython\cygrpc\completion_queue.pyx.pxi", line 61,
in grpc._cython.cygrpc._internal_latent_event
File "src\python\grpcio\grpc\cython\cygrpc\completion_queue.pyx.pxi", line 42,
in grpc._cython.cygrpc._next
KeyboardInterrupt
C
D:\分布式\experiment2>python client.py
C:\Users\85013\anaconda3\Lib\site-packages\google\protobuf\runtime_version.py:112: User
Warning: Protobuf gencode version 5.27.2 is older than the runtime version 5.28.2 at pu
bsub.proto. Please avoid checked-in Protobuf gencode that can be obsolete.
warnings.warn(
Server response: it's a message that only two clients can receive

C:\WINDOWS\system32\cmd x + v
File "C:\Users\85013\anaconda3\Lib\site-packages\grpc_channel.py", line 1181, in _c
all__
return _end_unary_response_blocking(state, call, False, None)
File "C:\Users\85013\anaconda3\Lib\site-packages\grpc_channel.py", line 1006, in _en
d_unary_response_blocking
raise _InactiveRpcError(state) # pytype: disable=not-instantiable
grpc._channel._InactiveRpcError: <InactiveRpcError of RPC that terminated with:
status = StatusCode.UNKNOWN
details = "Exception calling application: "
debug_error_string = "UNKNOWN:Error received from peer {created_time:"2024-10-
20T08:01:47.5447298+00:00", grpc_status:2, grpc_message:"Exception calling application:
">
D:\分布式\experiment2>
```

至此，消息订阅服务完成。

4. 遇到的问题及解决方法

在刚开始我的客户端代码为如下：

```
import grpc
from rpc.pubsub_pb2 import mes2client, mes2server
from rpc.pubsub_pb2_grpc import pubsubStub

# 定义一个函数来处理客户端逻辑
def run():
    # 配置通信的服务器IP地址和端口
    with grpc.insecure_channel('localhost:50000') as channel:
        # 创建客户端存根
        stub = pubsubStub(channel)
        # 向服务器发送消息，并接收服务器的响应
        response = stub.pubsubServe(mes2server(mes1='client'), timeout=500)
        # 打印服务器返回的消息
        print("Server response: " + response.mes2)

if __name__ == '__main__':
    run()
```

但这个代码只能一条一条接收消息，即服务器发送一条消息，客户端接收后即刻结束通信。

```
C:\WINDOWS\system32\cmd. x + v x
_for_termination
return _common.wait(
    """
File "C:\Users\85013\anaconda3\Lib\site-packages\grpc\_common.py", line 156, in wait
    _wait_once(wait_fn, MAXIMUM_WAIT_TIMEOUT, spin_cb)
File "C:\Users\85013\anaconda3\Lib\site-packages\grpc\_common.py", line 116, in _wait
    _once
    wait_fn(timeout=timeout)
File "C:\Users\85013\anaconda3\Lib\threading.py", line 655, in wait
    signaled = self._cond.wait(timeout)
File "C:\Users\85013\anaconda3\Lib\threading.py", line 359, in wait
    gotit = waiter._acquire(True, timeout)
    """
KeyboardInterrupt
^C
D:\分布式\experiment2>python server.py
C:\Users\85013\anaconda3\Lib\site-packages\google\protobuf\runtime_version.py:112: User
Warning: Protobuf gencode version 5.27.2 is older than the runtime version 5.28.2 at pu
bsub.proto. Please avoid checked-in Protobuf gencode that can be obsolete.
    warnings.warn(
mes:hello world!
mes:I'm gonna send you some mes

C:\WINDOWS\system32\cmd. x + v x
C:\Users\85013>D:
D:\>cd 分布式\experiment
系统找不到指定的路径。
D:\>cd 分布式
D:\分布式>cd experiment2
D:\分布式\experiment2>python client.py
C:\Users\85013\anaconda3\Lib\site-packages\google\protobuf\runtime_version.py:112: User
Warning: Protobuf gencode version 5.27.2 is older than the runtime version 5.28.2 at pu
bsub.proto. Please avoid checked-in Protobuf gencode that can be obsolete.
    warnings.warn(
Server response: hello world!
D:\分布式\experiment2>python client.py
C:\Users\85013\anaconda3\Lib\site-packages\google\protobuf\runtime_version.py:112: User
Warning: Protobuf gencode version 5.27.2 is older than the runtime version 5.28.2 at pu
bsub.proto. Please avoid checked-in Protobuf gencode that can be obsolete.
    warnings.warn(
Server response: I'm gonna send you some mes
D:\分布式\experiment2>

Microsoft Windows [版本 10.0.22631.4317]
(c) Microsoft Corporation. 保留所有权利。
C:\Users\85013>D:
D:\>cd 分布式\experiment2
D:\分布式\experiment2>python client.py
C:\Users\85013\anaconda3\Lib\site-packages\google\protobuf\runtime_version.py:112: User
Warning: Protobuf gencode version 5.27.2 is older than the runtime version 5.28.2 at pu
bsub.proto. Please avoid checked-in Protobuf gencode that can be obsolete.
    warnings.warn(
Server response: I'm gonna send you some mes
D:\分布式\experiment2>
```

因此我加入了个循环，即上面的代码，这样就能一直接收消息直至ctrl+C手动结束进程。