# 分布式系统_HW2

*21307289 刘森元*

# 1．理论题

## 1）分布式系统的软件体系结构有哪几种类型，各自有什么优缺点?

分布式系统的软件体系结构有以下几种类型：

1. 客户端-服务器架构：这是最常见的分布式系统架构，其中客户端应用程序通过网络与服务器进行通信。优点是易于扩展和维护，缺点是服务器成为单点故障。
2. 对等网络架构：在这种架构中，每个节点都是对等的，可以直接与其他节点通信。优点是去中心化和高度灵活性，缺点是安全性和一致性问题可能会变得更加复杂。
3. 三层架构：这种架构将系统分为表示层、业务逻辑层和数据层。表示层处理用户界面，业务逻辑层处理业务逻辑，数据层负责数据存储。优点是模块化和可维护性，缺点是增加了系统的复杂性。
4. 事件驱动架构：在这种架构中，系统通过事件的触发和处理进行通信和协调。优点是松散耦合和可扩展性，缺点是事件顺序和一致性的管理可能会变得复杂。
5. 微服务架构：这是一种将应用程序拆分为小型、自治的服务的架构。每个服务都可以独立开发、部署和扩展。优点是灵活性和可伸缩性，缺点是服务之间的通信和管理可能会变得复杂。

## 2）在点对点网络中，并不是每个节点都能成为超级对等节点，满足超级对等节点的合理要求是什么?

在点对点网络中，超级对等节点是指具有特殊功能或资源的节点，可以提供更多的服务或支持更多的连接。满足超级对等节点的合理要求通常包括以下几个方面：

1. 高性能和高带宽：超级对等节点需要具备较高的处理能力和网络带宽，以支持更多的连接和处理更多的数据流量。
2. 可靠性和稳定性：超级对等节点需要具备较高的可靠性和稳定性，能够持续提供服务而不易受到故障或攻击的影响。
3. 存储和计算能力：超级对等节点可能需要具备较大的存储空间和计算能力，以支持存储和处理大量的数据。
4. 网络可达性：超级对等节点需要具备良好的网络可达性，能够与其他节点建立可靠的连接，以便进行数据传输和通信。
5. 安全性和身份验证：超级对等节点需要具备较高的安全性，能够进行身份验证和数据加密，以保护节点和数据的安全。

需要注意的是，超级对等节点并非每个节点都能满足的要求，通常只有具备特定条件或资源的节点才能成为超级对等节点。

## 3）通过生成进程来构建并发服务器与使用多线程服务器相比有优点也有缺点。给出优点和缺点。

优点：

1. 高度稳定性：生成进程的主要优势之一是每个进程都是独立的，一个进程的崩溃不会影响其他进程的运行。这提高了服务器的稳定性和可靠性。

2. 容错能力强：由于每个进程都是独立的，一个进程的错误或异常不会影响其他进程的正常运行。这使得服务器具有更强的容错能力，即使一个进程出现问题，其他进程仍然可以继续运行。
3. 可扩展性：生成进程模型使得服务器能够在多个物理或虚拟机上运行，从而实现更好的可扩展性。每个进程可以在不同的计算资源上运行，从而提高了服务器的整体性能。

缺点：

1. 开销较大：相对于多线程服务器，生成进程需要更多的系统资源，例如内存和处理器。每个进程都需要独立的内存空间和上下文切换开销，这可能会导致服务器整体性能下降。
2. 通信成本高：由于生成进程之间的通信需要通过进程间通信（IPC）机制，例如管道或消息队列，因此通信成本相对较高。这可能会导致服务器在处理大量并发请求时的性能瓶颈。
3. 编程复杂性：与多线程服务器相比，生成进程模型的编程更加复杂。进程间通信和同步需要更多的代码和处理，这增加了开发和维护的复杂性。

## 4）维护到客户的TCP/IP链接的服务器是状态相关的还是状态无关的？说明理由。

维护到客户的TCP/IP连接的服务器通常是状态相关的。

服务器需要跟踪每个客户端连接的状态信息，包括连接的建立、终止、数据传输等。服务器必须维护关于每个连接的状态信息，例如客户端的IP地址、端口号、连接状态、数据缓冲区等。

1. 连接管理：服务器需要管理多个客户端连接，以便处理并发请求。服务器必须跟踪每个连接的状态，以便正确地处理请求和响应。
2. 数据传输：TCP/IP连接是全双工的，服务器和客户端可以在连接上进行双向的数据传输。服务器必须维护连接的状态以确保正确地接收和发送数据。
3. 错误处理：如果连接发生错误或异常，服务器需要根据连接的状态进行相应的错误处理。例如，服务器可能需要关闭连接、重新建立连接或发送错误消息给客户端。

## 5）代码迁移有哪些场景？为什么要进行代码迁移?

代码迁移（Code migration）可以发生在多种场景下：

1. 平台迁移：当应用程序需要从一个平台迁移到另一个平台时，例如从Windows迁移到Linux或从物理服务器迁移到云平台，代码迁移是必要的。这可能涉及到修改和调整代码以适应新的平台环境和要求。
2. 技术栈转换：当组织决定采用不同的技术栈或框架时，代码迁移是常见的。例如，从Java迁移到Python，或从Angular迁移到React等。这可能需要对现有代码进行重写或修改以适应新的技术栈。
3. 版本升级：当新版本的编程语言、框架或库发布时，为了获得新功能、修复漏洞或提高性能，代码迁移也是必要的。这可能涉及到代码的重写、调整或更新以适应新版本的要求和标准。
4. 代码重构：代码迁移也可以发生在进行代码重构的过程中。当代码存在质量问题、不符合最佳实践或难以维护时，进行代码迁移可以对代码进行重新组织、简化和优化。

代码迁移的目的通常是为了实现以下几个方面的需求：

1. 兼容性：通过代码迁移，可以使应用程序在新的平台、技术栈或版本下正常运行，确保兼容性和可用性。
2. 性能和效率：代码迁移可以帮助优化代码，提高性能和效率。新的平台、技术栈或版本可能提供更好的性能特性，通过迁移可以利用这些优势。
3. 维护和可扩展性：通过代码迁移，可以对代码进行重构和优化，使其更易于维护和扩展。新的技术栈或框架可能提供更好的工具和模式来支持代码的维护性和可扩展性。
4. 更新和安全性：代码迁移可以帮助应用程序更新到最新的版本，从而获得新功能、修复漏洞和提高安全性。

# 2. 实验题

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，并利用样例程序如Web程序等测试迁移过程中的性能损耗（如延迟变化或者请求错误率等）、观察发现，并撰写报告，报告的模板可以在课程网站找到。

```
apt update
apt upgrade
apt install docker docker.io
apt install criu

echo "{\"experimental\": true}" >> /etc/docker/daemon.json
systemctl restart docker

docker run -itd -v $PWD:/usr/src -w /usr/src --name looper python python
Hello_world.py
docker logs looper -f

docker checkpoint create looper ckp1
docker start looper --checkpoint=ckp1
```

## 实验环境

### Env-1

Macbook Pro 2021, Apple M1 Pro, Apple Silicon, ARM64/v8

macOS Ventura 13.5.2

UTM 4.2.0

Ubuntu-live-server 18.04 LTS, AMD64

Docker 20.10.21

CRIU 3.6

Python 2.7.17

### Env-2

Intel i5-13600KF, AMD64

Ubuntu-live-server 23.04

Docker 24.0.5

CRIU 3.17.1

Python 3.11.4

以 Macbook Pro 作为物理机，使用 UTM 进行虚拟机环境搭建。详见 UTM: 开源的多面手 macOS 虚拟机 - 知乎。配置好虚拟机后，使用 apt 进行必要环境安装。

```
apt update
apt upgrade
apt install docker docker.io
apt install criu
```

使用 `docker version && criu check` 检查，有

```
root@liusenyuan-21307289:~/src# docker version
Client:
 Version:           20.10.21
 API version:       1.41
 Go version:        go1.18.1
 Git commit:        20.10.21-0ubuntu1~18.04.3
 Built:             Thu Apr 27 05:50:21 2023
 OS/Arch:           linux/amd64
 Context:           default
 Experimental:      true

Server:
 Engine:
  Version:          20.10.21
  API version:      1.41 (minimum version 1.12)
  Go version:       go1.18.1
  Git commit:       20.10.21-0ubuntu1~18.04.3
  Built:            Thu Apr 27 05:36:22 2023
  OS/Arch:          linux/amd64
  Experimental:     true
 containerd:
  Version:          1.6.12-0ubuntu1~18.04.1
  GitCommit:
 runc:
  Version:          1.1.4-0ubuntu1~18.04.2
  GitCommit:
 docker-init:
  Version:          0.19.0
  GitCommit:
root@liusenyuan-21307289:~/src# criu check
Looks good.
```

可见已成功安装。

若要使用 `docker checkpoint` 等功能，需要开启 Docker 的实验性功能

```
echo "{\"experimental\": true}" >> /etc/docker/daemon.json
systemctl restart docker
```

使用 `docker info` 有

```
Client:
 Context:    default
 Debug Mode: false

Server:
 Containers: 0
  Running: 0
  Paused: 0
  Stopped: 0
 Images: 1
 Server Version: 20.10.21
 Storage Driver: overlay2
  Backing Filesystem: extfs
  Supports d_type: true
  Native Overlay Diff: true
  userxattr: false
 Logging Driver: json-file
 Cgroup Driver: cgroupfs
 Cgroup Version: 1
 Plugins:
  Volume: local
  Network: bridge host ipvlan macvlan null overlay
  Log: awslogs fluentd gcplogs gelf journald json-file local logentries splunk
syslog
 Swarm: inactive
 Runtimes: io.containerd.runc.v2 io.containerd.runtime.v1.linux runc
 Default Runtime: runc
 Init Binary: docker-init
 containerd version:
 runc version:
 init version:
 Security Options:
  apparmor
  seccomp
   Profile: default
 Kernel Version: 4.15.0-213-generic
 Operating System: Ubuntu 18.04.6 LTS
 OSType: linux
 Architecture: x86_64
 CPUs: 4
 Total Memory: 7.786GiB
 Name: liusenyuan-21307289
 ID: FZNE:W42B:XGHI:TE6Q:WI45:JCOZ:ZLTV:LXEQ:KVGW:N7TM:OFOF:WR3I
 Docker Root Dir: /var/lib/docker
 Debug Mode: false
 Registry: https://index.docker.io/v1/
 Labels:
```

```
 Experimental: true
 Insecure Registries:
  127.0.0.0/8
 Live Restore Enabled: false
```

可见 `Experimental: true` ，实验性功能已开启。

## 实验步骤

### 编写用例 Python 程序

可编写如下 Python 程序作为例子：

***Hello_world.py***

```python
from time import sleep
from sys import stdout

count = 0
id = '21307289'

while True:
    print('%04d -> %s' % (count, id))
    count += 1
    sleep(1)
    stdout.flush()
```

程序会每隔一秒输出一次结果，运行效果如下：

```
root@liusenyuan-21307289:~/src# python Hello_world.py
0000 -> 21307289
0001 -> 21307289
0002 -> 21307289
0003 -> 21307289
0004 -> 21307289
0005 -> 21307289
```

### 创建容器并运行

使用如下命令创建名为 looper 的容器运行 Python 程序：

```
docker run -itd -v $PWD:/usr/src -w /usr/src --name looper python python
Hello_world.py
```

使用 `docker logs looper -f` 查看有

```
root@liusenyuan-21307289:~/src# docker logs looper -f
0000 → 21307289
0001 → 21307289
0002 → 21307289
0003 → 21307289
0004 → 21307289
0005 → 21307289
0006 → 21307289
0007 → 21307289
0008 → 21307289
0009 → 21307289
0010 → 21307289
0011 → 21307289
```

可见容器成功运行了 ***Hello_world.py***

### 创建 checkpoint

在此前运行容器的基础上创建 checkpoint

```
docker checkpoint create looper ckp1
```

创建后使用 `docker logs looper -f` 查看

```
...
0110 → 21307289
0111 → 21307289
0112 → 21307289
0113 → 21307289
0114 → 21307289
```

程序停止运行，0114 为终止号

### 转移 checkpoint

使用 `scp <local_file> <remote_username>@<remote_ip>:<folder>` 传输 checkpoint 文件到 Env-2

### 从 checkpoint 恢复容器运行

首先创建新容器

```
docker run -itd -v $PWD:/usr/src -w /usr/src --name looper python
```

并从 checkpoint 恢复运行

```
docker start looper --checkpoint=ckp1
```

使用 `docker logs looper -f` 查看有

```
...
0115 → 21307289
0116 → 21307289
0117 → 21307289
0118 → 21307289
0119 → 21307289
0120 → 21307289
```

程序从编号 0115 执行，而非从 0001 重新开始，迁移成功。