Topic 4: Microservice Architecture: Services and Overall Control Plane Implementation

Nov. 18, 2021

Y. Richard Yang
Computer Science Department, Yale University
Email: yry@cs.yale.edu

http://zoo.cs.yale.edu/classes/cs434/
Final Project planning
- Nov. 18: Iteration of topics
- Dec. 3: Checkpoint (2-page update)
- Dec. 22: Due (report)

Remaining lectures planning
Recap: High Level SDN Programming

- Basic goal: simplify the programming of distributed datapath
  - The Maple algorithmic SDN programming model
    - Trace tree: tracks decision dependency on input, and converts the tree to datapath, for general purpose language
    - Unified dependency to handle environment changes
    - Extend to stream attributes for L2-L7 programming
    - Embed declarative language (e.g., route algebra)
Recap: Extend OpenFlow 1.x: Programming Protocol-independent Packet Processors (P4)

- Support reconfigurability, protocol independence, and target independence
Outline

- Admin and recap
- Microservice based architecture
Motivation

Many applications supported by network systems of interests (e.g. cloud) are based on the microservices arch

Many network systems are designed based on the microservices architecture
Issues to Think About

Look into service mesh (sidecar) as joint network-app design

Understand design and implementation of architecture realizing microservices apps

Understand the issues of virtualization for network systems

Analyze and apply microservice architecture to network systems control plane
Roadmap

- **Today**
  - Overview and background
  - Microservices architecture services (Kubernetes)
  - Microservices architecture overall design and implementation

- **Tuesday after the break**
  - Microservices architecture interconnect (network system) design and implementation
Outline

- Admin and recap
- Microservice based architecture
  - Overview and background
An Example (Micro)Service Application

- A toy example from
  - Slides: https://qconuk2019.container.training/#35
  - Git: git clone https://github.com/jpetazzo/container.training
  - %cd dockercoins

- Function: a coin mining farm
  - generate random bytes
  - hash these bytes and check if a hit
  - monitor and display progress

- For modular design, divide into multiple functions (services) and each service implemented in its own platform (e.g., ruby, python, js)
Example App Services

- 5 components (services)
  - redis (keep track of states e.g., counter, found coins)
  - rng, hasher, worker work on mining
    - worker is the main orchestrator
      - invokes web service rng to generate random bytes
      - invokes web service hasher to hash these bytes
      - every second, worker updates redis to indicate how many loops were done
  - webui queries redis, and computes and exposes "hashing speed" in the browser
Distributed Services
Service Installation and Programming: rng

- Install python
- pip install Flask
- cp rng.py /
- python rng.py

Demo: VirtualBox VM install

```python
from flask import Flask, Response
import os
import socket
import time

class App:
    def __init__(self):
        self.app = Flask(__name__)

    def run(self):
        self.app.run(host="0.0.0.0", port=80)

app = App()
app.run()
```

rng.py
Service Installation and Programming: hasher

- Install ruby
- apk add --update build-base curl
- gem install sinatra
- gem install thin
- cp hasher.rb /
- ruby hasher.rb

```ruby
require 'digest'
require 'sinatra'
require 'socket'

set :bind, '0.0.0.0'
set :port, 80

post '/do'
  # Simulate a bit of delay
  sleep 0.1
  content_type 'text/plain'
  "#{Digest::SHA2.new().update(request.body.read)}"
end

get '/' do
  "HASHER running on #{Socket.gethostname}\n"
end
```
Service Installation and Programming: worker

- Install python
- pip install redis
- pip install requests
- cp worker.py /
- python worker.py

```python
import logging
import os
from redis import Redis
import requests
import time

DEBUG = os.environ.get("DEBUG", "").lower().startswith("y")

log = logging.getLogger(__name__)
if DEBUG:
    log.basicConfig(level=logging.DEBUG)
else:
    log.basicConfig(level=logging.INFO)
    log.getLogger("requests").setLevel(logging.WARNING)

redis = Redis("redis")

def get_random_bytes():
    r = requests.get("http://rng/32")
    return r.content

def hash_bytes(data):
    r = requests.post("http://hasher/",
                      data=data,
                      headers={"Content-Type": "application/octet-stream"})
    hex_hash = r.text
    return hex_hash
```
```python
def work_loop(interval=1):
    deadline = 0
    loops_done = 0
    while True:
        if time.time() > deadline:
            log.info(f"{} units of work done, updating hash counter".format(loops_done))
            redis.incrby("hashes", loops_done)
            loops_done = 0
            deadline = time.time() + interval
            work_once()
            loops_done += 1

def work_once():
    log.debug("Doing one unit of work")
    time.sleep(0.1)
    random_bytes = get_random_bytes()
    hex_hash = hash_bytes(random_bytes)
    if not hex_hash.startswith('0'):
        log.debug("No coin found")
        return
    log.info("Coin found: {}...".format(hex_hash[:8]))
    created = redis.hset("wallet", hex_hash, random_bytes)
    if not created:
        log.info("We already had that coin")

if __name__ == "__main__":
    while True:
        try:
            work_loop()
        except:
            log.exception("In work loop:")
            log.error("Waiting 10s and restarting.")
            time.sleep(10)
```

worker.py
Service Installation and Programming: webui.js

- Install vm w/ node
- npm install express
- npm install redis
- cp files/ /files/
- cp webui.js /
- node webui.js
Discussions

- Benefits of service based architecture

- Downside of VM based service based architecture
Instead of using virtual machines, use lighter-weight isolation mechanism (aka microservices)

One main isolation mechanism: container

Service based and microservice based architectures share many common issues
Background: Namespace

- Container is just a higher abstraction built on top of Linux namespace and cgroup for isolation
  - Simple analogy: VM is like build a new house, and a namespace just carves an isolation space
  - Good references: Docker using cgroups, namespace and beyond [https://www.youtube.com/watch?v=sK5i-N34im8](https://www.youtube.com/watch?v=sK5i-N34im8)

- In Linux, each process has its set of namespaces

- Demo: show namespaces of a process
  - $ ls -l /proc/<pid>/ns

```bash
$ ls -l /proc/$$/ns
  total 0
  lrwxrwxrwx. 1 dj dj 0 Jan 4 04:12 ipc -> ipc:[4026631839]
lrwxrwxrwx. 1 dj dj 0 Jan 4 04:12 mnt -> mnt:[4026631840]
lrwxrwxrwx. 1 dj dj 0 Jan 4 04:12 net -> net:[4026631956]
lrwxrwxrwx. 1 dj dj 0 Jan 4 04:12 pid -> pid:[4026631836]
lrwxrwxrwx. 1 dj dj 0 Jan 4 04:12 user -> user:[4026631837]
lrwxrwxrwx. 1 dj dj 0 Jan 4 04:12 uts -> uts:[4026631838]
```
Namespace System Calls

- Three system calls to create process and control namespaces

```c
int clone(int (*child_func)(void *), void *child_stack, int flags, void *arg);

int setns(int fd, int nstype);

int unshare(int flags);
```

See [https://man7.org/linux/man-pages/man2/clone.2.html](https://man7.org/linux/man-pages/man2/clone.2.html)
Namespace support since 2.6.24
```c
#define _GNU_SOURCE
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <unistd.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>
#include <iostream.h>

#define STACK_SIZE (1024 * 1024)

// sync primitive
int checkpoint[2];

static char child_stack[STACK_SIZE];
char* const child_args[] = { 
  "/bin/bash",
  NULL
};

int child_main(void* arg)
{
  char c;

  // init sync primitive
  close(checkpoint[1]);

  // setup hostname
  printf(" - [%5d] Hello ?\n", getpid());
  sethostname("In Namespace", 12);

  // remount "/proc" to get accurate "top" & "ps" output
  mount("/proc", "/proc", "proc", 0, NULL);

  // wait for network setup in parent
  read(checkpoint[0], &c, 1);

  // setup network
  system("ip link set lo up");
  system("ip link set veth1 up");
  system("ip addr add 169.254.1.2/30 dev veth1");

  execv(child_args[0], child_args);
  printf("Doops\n");
  return 1;
}

int main()
{
  // init sync primitive
  pipe(checkpoint);

  printf(" - [%5d] Hello ?\n", getpid());

  int child_pid = clone(child_main, child_stack+STACK_SIZE,
                        CLONE_NEWUTS | CLONE_NEWIPC | CLONE_NEWPID | CLONE_NEWNS | CLONE_NEWNET | SIGCHLD, NULL);

  // further init: create a veth pair
  char* cmd;
  asprintf(&cmd, "ip link set veth1 netns %d", child_pid);
  system("ip link add veth0 type veth peer name veth1");
  system(cmd);
  system("ip link set veth0 up");
  system("ip addr add 169.254.1.1/30 dev veth0");
  free(cmd);

  // signal "done"
  close(checkpoint[1]);

  waitpid(child_pid, NULL, 0);
  return 0;
}
```
Network Namespace

- Discussion: Which network resources will be labelled with network namespace?
  - Linux network namespace
    - Devices
    - Routing tables
    - Netfilters
    - Networking stack (sockets, ...)
- References (more in next class)
  - https://www.youtube.com/watch?v=6v_BDHIGOY8
**Docker Demo**

- Docker is widely used and conceptually just a tool to start services and put them into namespaces (containers)

**Demo:**
- see dockercoin source code
- Execute dockers
  - Start all dockers
    - %docker-compose up
  - See dockers running (a separate terminal)
    - %docker-compose ps
  - See webui results
    - http://0.0.0.8000/index.html
  - Shutdown all containers
    - %docker-compose down
Example Operations in a (Micro)services Architecture

- Start 10 containers using image hasher:v1.3
- Place an internal load balancer in front of these containers
- Start 3 containers using image webui:v1.3
- Place a public load balancer in front of these containers
- It's Black Friday, traffic spikes, grow our cluster and add containers
- New release! Replace containers with the new image webu:v1.4
- Keep processing requests during the upgrade; update containers one at a time
Example Services in a Microservices Architecture

- Basic autoscaling
- Blue/green deployment, canary deployment
- Long running services, but also batch (one-off) jobs
- Overcommit our cluster and *evict* low-priority jobs
- Run services with *stateful* data (databases etc.)
- Fine-grained access control defining *what* can be done by *whom* on *which* resources
- Integrating third party services (*service catalog*)
- Automating complex tasks (*operators*)
Outline

- Admin and recap
- Microservice based architecture
  - Overview and background
  - Microservices architecture services (Kubernetes)
Kubernetes Basic Concepts

- Containers
- Workloads
  - Pods
  - Workload resources: Deployment, ReplicaSet, DaemonSet ...
- Services, load balancing and networking
  - Service, topology-aware traffic routing with topology keys, DNS for services and pods, ...
- Storage
- Scheduling
- Configuration
pods

- A pod (as in a pod of whales or pea pod) is the smallest deployable units of computing that one can create and manage in Kubernetes.
  - https://kubernetes.io/docs/concepts/workloads/pods/

- A Pod models an application-specific "logical host": it contains one or more application containers which are relatively tightly coupled

- Pod's contents are always co-located and co-scheduled, and run in a shared storage and network context.
Pods (Offline)

- Pods in a Kubernetes cluster are used in two main ways:
- **Pods that run a single container.** The "one-container-per-Pod" model is the most common Kubernetes use case; in this case, you can think of a Pod as a wrapper around a single container; Kubernetes manages Pods rather than managing the containers directly.
- **Pods that run multiple containers that need to work together.** A Pod can encapsulate an application composed of multiple co-located containers that are tightly coupled and need to share resources. These co-located containers form a single cohesive unit of service—for example, one container serving data stored in a shared volume to the public, while a separate *sidecar* container refreshes or updates those files. The Pod wraps these containers, storage resources, and an ephemeral network identity together as a single unit.
Node, Container, Pod, K8s Requirements

- Pods are allocated IP addresses from a given range
- IP-Per-Pod: Each Pod has a single IP from the range
- Pods run in a direct-connect, flat address space
  - Containers in the same Pod share IP, same Linux network stack, and can connect using localhost
Demo (Start a pod)

- Start a single pod that consists of a single container using command line
  - // check existing pods
    kubectl get all
  - // watch pods set
    kubectl get pod -w
  - // start a pod using container alpine, ping 1.1.1.1 DNS server
    kubectl run pingpong --image alpine ping 1.1.1.1
  - kubectl logs pod/pingpong --tail 1 --follow
  - kubectl delete pod/pingpong // remove the pod
(Offline) Background

- kubectl run pingpong --image alpine calls docker run alpine
- If the Engine needs to pull the alpine image, it expands it into library/alpine
- library/alpine is expanded into index.docker.io/library/alpine
- To use some other registry than index.docker.io, specify it in the image name, e.g.,
  - docker pull gcr.io/google-containers/alpine-with-bash:1.0
Exercise (start a pod using .yaml)

- Starting a pod using a .yaml file (using a job)
  - kubectl apply -f pod-echo.yaml
  - Kubectl get all // see if started
  - Kubectl logs pod/... // see outcome
  - Kubectl delete pod/pingpong
Outline

- Admin and recap
- Microservice based architecture
  - Overview and background
  - Microservices architecture services (Kubernetes)
    - pod
    - deployment
Deployment and ReplicaSet

- A deployment provides a declarative specification about a desired state, and the Deployment Controller changes the actual state to the desired state at a controlled rate.

- A deployment is a high-level construct
  - allows scaling, rolling updates, rollbacks
  - multiple deployments can be used together to implement a canary deployment
  - delegates pods management to replica sets

- A replica set is a low-level construct
  - makes sure that a given number of identical pods are running
  - allows scaling
  - rarely used directly

- A replication controller is the (deprecated) predecessor of a replica set
Demo: Deployment

- Start a deployment of http server
  - // window 1: watch pods being created
    - `kubectl get pod -w`
  - // window 2, // a web server image called nginx
    - `kubectl create deployment ngservers --image=nginx`
    - `kubectl scale deployment ngservers --replicas=3`
    - `kubectl get all`

- Deployment management
  - // Kill one pod and see the reaction
    - `kubectl delete pod/ngservers-xxxx`
    - `kubectl delete deploy/ngservers`
Outline

- Admin and recap
- Microservice based architecture
  - Overview and background
  - Microservices architecture services (Kubernetes)
    - pod
    - deployment
    - services
A service is a stable address for a pod (or a bunch of pods)
- If you want to connect to the pod(s), you need to create a service
- Once a service is created, CoreDNS can resolve it by name (i.e. after creating service hello, the name hello will resolve to something)

There are different types of services
- ClusterIP, NodePort, LoadBalancer, ExternalName
Glimpse of Complexity (for next class)
Services

- **ClusterIP (default type)**
  - a virtual IP address is allocated for the service (in an internal, private range)
  - this IP address is reachable only from within the cluster (nodes and pods)
  - your code can connect to the service using the original port number

- **NodePort**
  - a port is allocated for the service (by default, in the 30000-32768 range)
  - that port is made available on all the nodes and anybody can connect to it
  - the code must be changed to connect to that new port number
Services

- **LoadBalancer**
  - an external load balancer is allocated for the service
  - the load balancer is configured accordingly (e.g.: a NodePort service is created, and the load balancer sends traffic to that port)
  - available only when the underlying infrastructure provides some "load balancer as a service" (e.g. AWS, Azure, GCE, OpenStack...)

- **ExternalName**
  - the DNS entry managed by CoreDNS will just be a CNAME to a provided record
  - no port, no IP address, no nothing else is allocated
Demo

- Expose the HTTP port of our server:
  - `kubectl expose deployment ngservers --type=NodePort --port 8888`
- Check service created
  - `kubectl get all`
- Or look up which IP address was allocated:
  - `kubectl get service`
- Check the endpoints of the service
  - `kubectl describe service ngservers`
  - `kubectl describe endpoints ngservers`
  - `kubectl get pods -l app=ngservers -o wide`
(Offline) Try the DockerCoin App Using Kubernetes

- // the dockers are already created
dockercoins/<name>:v0.1
  see tutorial on creating docker images and host at registry

- kubectl create deployment redis --image=redis
- kubectl create deployment worker
  --image=dockercoins/worker:v0.1
- Same for hasher rng webui worker
Try the DockerCoin App Using Kubernetes

- `kubectl logs deploy/rng`
- `kubectl logs deploy/worker`

Error (not expose)
- `kubectl expose deployment redis --port 6379`
- `kubectl expose deployment rng --port 80`
- `kubectl expose deployment hasher --port 80`
(Offline) Try the DockerCoin App Using Kubernetes

- Create a NodePort service for the Web UI:
  - kubectl expose deploy/webui --type=NodePort --port=80
- Check the port that was allocated:
  - kubectl get svc
- See webui
  - http://0.0.0.0:31207/index.html
Pod: models an application-specific "logical host"
  - A pod contains one or more application containers which are relatively tightly coupled
  - Pod's contents are always co-located and co-scheduled, and run in a shared storage and network context

Deployment: provides a declarative specification about a desired state
  - Deployment controller changes the actual state to the desired state at a controlled rate
    - kubectl create deployment nginx --image=nginx
    - kubectl scale deployment nginx --replicas=3

Service: provides a stable address for a pod (or a bunch of pods)
  - ClusterIP, NodePort, LoadBalancer, ExternalName
    - kubectl expose deployment nginx --port 8888
    - kubectl expose deploy/webui --type=NodePort --port=80 // assume allocate port 31207
    - http://0.0.0.0:31207/index.html
Outline

- Admin and recap
- Microservice based architecture
  - Overview and background
  - Microservices architecture services (Kubernetes)
  - Microservices architecture overall design and implementation
Microservices Architecture
Implementation Setting

- Typically in a cluster (cloud, edge cloud)

Exercise: How may you design the control system of a microservice architecture (what components, what they do)?
  - Consider reliability, scalability, extensibility
Kubernetes Implementation Architecture Design

Exercise: See any services used in its own implementation?

%kubectl -n kube-system get all
Kubernetes Cluster

- `kubectl cluster-info`
  - Kubernetes master is running at https://kubernetes.docker.internal:6443
    - MacOS
dinamevalueutil -q host -a name kubernetes.docker.internal
  - KubeDNS is running at https://kubernetes.docker.internal:6443/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy
Controller-Manager+Scheduler+Etcd Work-Flow Picture

1. Create ReplicaSet
2. Create ReplicaSet
3. Notify create Replica Set event
4. Notify create Replica Set event
5. Create Pod
6. Create Pod
7. Notify create Pod event
8. Notify create Pod event
9. Modify Pod
10. Modify Pod
11. Notify modify Pod event
12. Notify modify Pod event
5gc CP
Controller Managers

- Replication Controller
- Node Controller
- ResourceQuota Controller
- Namespace Controller
- ServiceAccount Controller
- Token Controller
- Service Controller
- Endpoint Controller
Example: Endpoint Controller
Exercise: Scheduler

- Function of scheduler: match node to pod
- Exercise: How may you design the scheduler?
Kubernetes Scheduler Design
Scheduler Predicates

- NoDiskConflict
- PodFitsResources
- PodSelectorMatches
- PodFitsHost
- CheckNodeLabelPresence
- CheckServiceAffinity
- PodFitsPorts

- Default loaded
  - PodFitsPorts(PodFitsPorts)
  - PodFitsResources(PodFitsResources)
  - NoDiskConflict(NoDiskConflict)
  - MatchNodeSelector(PodSelectorMatches)
  - HostName(PodFitsHost)
Scheduler Ranking Algorithms

- **LeastRequestedPriority:**
  - score\(=\)\(\text{int}\left(\frac{(\text{nodeCpuCapacity} - \text{totalMilliCPU})\times10}{\text{nodeCpuCapacity}}\right.\)\(+\left.\frac{((\text{nodeMemoryCapacity} - \text{totalMemory})\times10)}{\text{nodeCpuMemory}}\right)/2\)

- **CalculateNodeLabelPriority:**

- **BalancedResourceAllocation:**
  - score\(=\)\(\text{int}(10 - \text{math.abs}(\text{totalMilliCPU}/\text{nodeCpuCapacity} - \text{totalMemory}/\text{nodeMemoryCapacity})\times10)\)