华中科技大学

课程实验报告

课程名称：数据中心技术

指导老师：施展

姓名：刘建宇

学号：M201773202

日期：2017.11.29

一、实验目的

解决网络性能问题是一个有挑战性的任务，特别是在大规模数据中心网络中检测网络故障。Detector 是一个网络监视系统，该系统可以在用较少的资源在很短时间时间内精确检测和定位网络故障。Detector 通过将侦查、定位和选择侦查路径完成这项任务，所以数据包丢失能够不用额外的工具帮助，只通过端到端的观测值来进行定位。

本实验使用 detector 对数据中心网络进行故障监测。仿真模拟 detector 的伸缩性、可行性和效率。

二、实验内容

Detector 工作是一个三步循环：路经计算，网络探测和丢失定位。

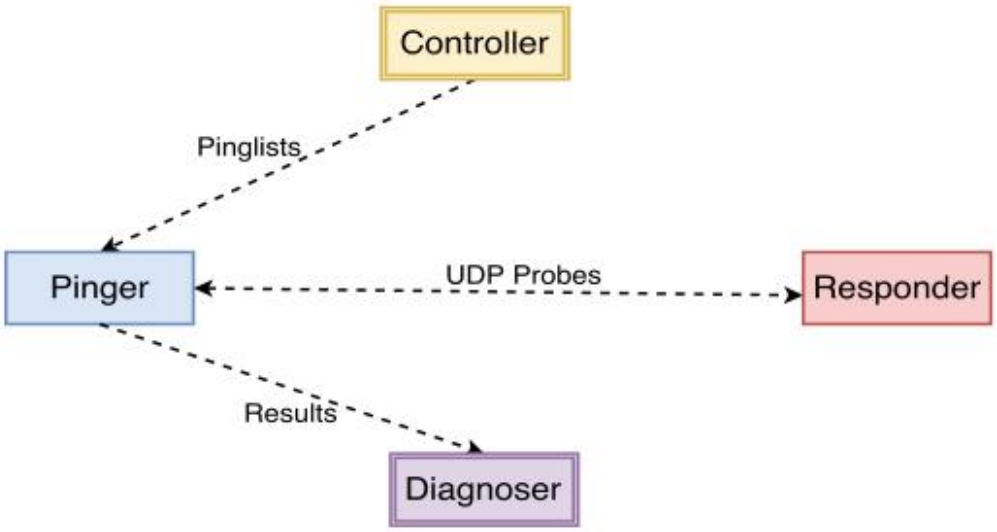
路经计算：在每个循环的开始，控制器读取数据中心拓扑结构和数据中心管理服务中的服务器状况。控制器然后选择每个机架顶端交换机中的发射器，构建和向它们分发 ping 清单。

网络探测：接下来，探针包被沿着特定的路线在 DCN 中发送。因为数据中心通常采用 ECMP 来进行负载平衡，我们必须使用源路由来控制每个探针包经历的路径，这可以用多种方法来实现。一个一般可行的解决方法是使用数据包的封装和解封装来创建端到端通道，尽管这会导致数据包在 VXLAN 或 NVGRE 创建的虚拟网络中的二次封装。以图 1 的 fattree 网络为例：固定核心交换机，在两个 inter-pod 服务器之间只有唯一的一条路径；我们可以使用移动 IP 数据封装和隧道来包装一个服务器上的探针；数据包到达核心路由器之后，外头部被移除，数据包被路由到真正的目的地。这样一个源路由选择机制只在服务器和核心路由器上造成了少量开支。

丢失定位：探针丢失测量由丢失定位算法在诊断器进行聚合和分析。我们精确定位故障链接，评估丢失率，向网络操作者发送后续操作（例如检查交换机日志）的警告。

三、实验原理

Detector 包括四个松耦合部分：一个控制器，一个诊断器，一个发射器和一个反应器，如下图所示



控制器：逻辑控制器定期的构建探针矩阵来指明发送探针的路径。我们主要关注相连路由器之间连接的故障定位，因为连接服务器和机架顶端交换机 ToR 的连接上的故障可以在下一段讨论的内容中很容易被识别出来。探针矩阵指明了机架顶端交换机 ToR 之间的路径。因此我们不依赖 ToR 的网络反应速度能力，探针由 2-4 个已选的在 ToR 下的服务器（这就是声波发射器）发送。

发射器：每一个发射器都收到了来自控制器的 ping 发射清单，清单包括了目标服务器，探针格式和 ping 配置。由 ToR 交换机到不同目的地的探针路径分布在该 ToR 交换机下的服务器中的 ping 清单里，为了保证一定的容错率，每条路径至少分发到两个发射器中。这样，万一某一个发射器宕机，其他同机架的发射器也可以探测到该路径，避免了任何连接覆盖中的大落差。为了侦测服务器和各自机架顶端交换机之间连接的故障，发射器也探测同机架下的其他服务器。即便是大数据中心网络，每个发射器的探针路径的数量不会超过一百。探针包通过 UDP 发送。虽然在 DCN 中主要使用 TCP 进行传输，但 DCN 在绝大多数情况下不会区别对待 TCP 和 UDP 传输，因此 UDP 探针也能够显示网络情况。当一个发射器侦测到一个探针丢失时，它能够通过发送两个同样内容的探针包来确定丢失模式。

反应器：反应器是一个轻量级的组件运行在所有的服务器上。根据收到的探针包，反应器发出回应。一个反应器不维持任何状态，所有的探测结果都由发射器记录。

诊断器：每个发射器记录丢包信息并发送到诊断器来做丢失定位。这些日志信息被保存到数据库中做实时的分析和后面的查询。诊断器运行 PLL 算法来精确定位包丢失和评估问题链接的损耗率。

为了控制器和诊断器的容错性和可扩展性，我们使用了已有的解决方法（例如软件负载平衡）

四、实验器材与实验环境

操作系统：Windows 7 x64

安装内存：4GB

处理器：Intel(R)Core(TM) i5-3317U CPU @ 1.70GHz 1.70GHz

编译环境：Python 2.7.6

五、实验步骤

在大型服务器（或者控制器能够在能够处理大规模网络的多台分布式服务器）上运行控制器，同时服务器上还运行了一个监视器，来监视其他服务器的健康状况，移除坏的服务器。控制器运行 PMC 算法每隔 10 分钟重新计算探针矩阵，基于来自监视器服务的当前的网络拓扑结构。计算得到的探针矩阵被分割到 XML 格式 ping 清单文件来分配到发射器去。一个 ping 清单文件包含文件版本信息，发射器的 IP 地址，反应器的 IP 地址，传输端口号，包发送间隔和核心交换机的 IP 地址。我们的测试显示，控制器在最大化带宽使用一个核心消耗 688.56Mb/s 平均每秒可以操作 4473 个 ping 清单。因为发射器部署在很少量的服务器上（大约服务器总数的 10%），当发射器每轮需要 ping 清单的时间稍微随机化一点，控制器便能够支持超过 10W 发射器。

每个发射器执行着一个通讯模块和一个探测模块。通讯模块负责与控制器和诊断器的连接通信。它通过每轮（10 分钟）的 HTTP GET 请求从控制器获取 ping 清单文件。探测模块根据 ping 清单产生探测包，并且将它们通过 IP-in-IP 封装。在我们的实验中，一个发射器循环遍历了一条路径个一系列端口，在每个端口发出了一些包。每个探针包平均大小为 850bytes，

搭载了一个特殊的 DSCP 值在 IP 头部中，以测试不同的 QoS 类。如果在 100ms 内没有收到回应的话，我们标记其为丢失。一个发射器通过循环遍历 ping 清单中的路径多次重复发送数据包，以每秒是 10 个包的频率。每 30 秒，发射器汇总一次探测结果（丢包的数量和发送到每条探测路径上数据包的数量）到一个 XML 文件中并用一个 HTTP POST 请求将其发送到诊断器中。反应器模块运行在所有服务器的用户空间中，它坚挺着一个特殊端口，当数据包到达时，它在上面加上一个时间戳然后将数据包发回。发射器和反应器在服务器上只产生了很少的开支。（6.3 中可以看到）

诊断器是一个 web 服务器模块，运行在与控制器同一服务器上。它运行 PLL 算法来每半分钟进行一次故障定位（使用过去 30 秒收到的探测结果）。给出了我们试验台的服务器限制数量之后，我们运行一个虚拟机来仿真一个服务器。

创建一个 4 进制 fattree 试验台,包括有 20 个 ONet 交换机，每个都配备有基于 FPGA（可编程）硬件可配置数据平面。4 个 1GB 以太网端口和一个专用的管理端口。因此在 detector 中我们不需要可编程的交换机，使用 SDN（自防御）交换机帮助我们扩展了在真实数据中心网络中的故障的种类的测评。特别的，我们将所有的故障分为三种类型：

满包丢失 在安装时将 OpenFlow 设置高优先级，为了终止所有来自于特定端口的数据包，以此来评估一个有满包丢失的故障链接。为了测试一个交换机宕机的情况，我们设置了一套规则就是在交换机上终止所有的数据包。

确定性的局部丢失 有明显特征的数据包（有特定的 IP，端口号）会在一条确切的链接上丢失，比如数据包黑洞或者配置丢失路由规则。为了评估这样的故障，我们在交换机上安装规则来匹配和移除某些数据包的头部。

随机局部丢失 有时一条链接上的数据包被随机的移除了，比如由于位翻转、CRC 故障、缓冲区溢出等等。SDN 交换机不支持随机的数据包丢失。为了评估这样的故障，我们在交换机上安装一些规则来向一个已评估的坏掉的连接到 SDN 控制器的链接重新递送所有数据包，并且 SDN 控制器以某个概率移除这些数据包，遵循从中提取的模式。

由于在真实世界的数据中心中无法得到丢失的数据，我们根据故障测试和传输测试。同时还设置了几个参数，例如链接和交换机故障百分比，链接故障率，不同层中交换机的故障率，所有的都基于以上的测试。

Controller.py

```
import subprocess
import threading
import time
import sys
import globvar
import consprobmat

# run http server
def http():
    logger.info('Starting http server...')
    cmd = 'python -m SimpleHTTPServer ' + str(globvar.controller_listen_port)
    subprocess.call(cmd, shell= True, cwd='./')

# compute probe path periodically if topology changes and generate pinglist for each server
def gen_lists():
```

```
    k = 4        # 4-ary Fattree
    numPod = 4    # compute paths for all 4 pods
    ide = 1            # identifiability, at most 1 for 4-ary Fattree
    cov = 3            # coverage
    coef = 1   # coefficient for score computation
    cores = 1  # parallel computation
    choice = globvar.sys_choice

    while(True):
        # compute probe matrix
        logger.info('compute probe matrix...')
        if choice == 1:         # deTector
            consprobmat.consprobmat(k, numPod, ide, cov, coef, cores)
        time.sleep(globvar.time_cyc)

def main():
    global logger

    globvar.init()
    logger = globvar.logger

    # create two threads, one for sending pkts and another for receiving pkts
    thrd_path = threading.Thread(name='path', target=gen_lists)
    thrd_path.setDaemon(True)
    thrd_path.start()

    time.sleep(1)

    thrd_http = threading.Thread(name='http', target=http)
    thrd_http.setDaemon(True)   # the child thread will forcefully exit if parent exits
    thrd_http.start()

    while(True):
        time.sleep(3)

if __name__ == "__main__":
        main()
```

diagnoser.py

```
import time
import threading
import sys
import globvar
import recvres
```

```
def main(argv):
	globvar.init(argv)

	logger = globvar.logger
	result_dir = globvar.result_dir
	# receive result => preprocessing result => fault localization
	thd_recv_result = threading.Thread(name='recv_result', target= recvres.recv_result,
args=(result_dir, ))
	thd_recv_result.setDaemon(True)
	thd_recv_result.start()

	while(True):
		time.sleep(10)

if __name__ == "__main__":
	if len(sys.argv) != 3:
			print "Please input controller ip and controller port!"
		print "Usage Example: python diagnoser.py 100.1.0.1 8180"
			sys.exit(1)
	main(sys.argv)
```

pinger.py

```
import time
import threading
import sys
import os
import subprocess
from datetime import datetime

import globvar
import updlist
import uping
import netbouncer
import fbtracert

cpus = []
mems = []

# monitor system cpu and mem usage
def monitor(fN):
	global cpus, mems

	pid = os.getpid()
```

```
        cmd = ['ps', '-p', str(pid), '-o', 'pcpu=', '-o', 'rssize=']
        #0.0    12252
        output = subprocess.check_output(cmd)
        output = output.split('\n')[0].split(' ')
        cpu_usage = -1
        mem_usage = -1
        try:
            cpu_usage = float(output[1])
            mem_usage = int(output[2])/1024.0
        except:
            globvar.logger.error('Getting CPU and memory usage error - index out of range!')
            cpu_usage = float(output[0])
            mem_usage = int(output[1])/1024.0

        if cpu_usage >= 0 and mem_usage >= 0:
            cpus.append(cpu_usage)
            mems.append(mem_usage)

        # batch write
        if len(cpus) == 20:    # 60 seconds
            fh = open(fN, 'a')
            for i in range(len(cpus)):
                fh.write(str(cpus[i]) + ' ' + str(mems[i]) + '\n')
            fh.close()
            cpus = []
            mems = []

def main(argv):

        globvar.init(argv)

        # update pinglist => udp ping => send logs
        # create two threads, one for updating pinglist and another for probes
        thrd_updlist = threading.Thread(name='updlist', target=updlist.update_pinglist)
        thrd_updlist.setDaemon(True)       # the child thread will forcefully exit if parent exits
        thrd_updlist.start()

        thrd_probe = threading.Thread(name='uping', target=uping.uping)
        thrd_probe.setDaemon(True)
        thrd_probe.start()

        # collect memory and cpu usage info
        date_time = datetime.today()
        timestr = str(date_time.year) + '-' + str(date_time.month) + '-' + str(date_time.day) + '-' +
```

```
str(date_time.hour) + ':' + str(date_time.minute) +    ':' + str(date_time.second)
        fN = './cpu_mem/pinger_cpu_mem.' + timestr + '.log'
        while(True):
                #monitor(fN)
                time.sleep(3)

if __name__ == "__main__":
        if len(sys.argv) != 4:
                        print "Please input host ip, controller ip and controller port!"
                print "Usage example: python pinger.py 10.0.0.2 100.1.0.1 8080"
                        sys.exit(1)
        main(sys.argv)
```

responder.py

```
import os
import subprocess
import threading
import time
import sys
from datetime import datetime

import globvar
import upong


cpus = []
mems = []

def monitor(fN):
        global cpus, mems

        pid = os.getpid()
        cmd = ['ps', '-p', str(pid), '-o', 'pcpu=', '-o', 'rssize=']
        #0.0    12252
        output = subprocess.check_output(cmd)
        output = output.split('\n')[0].split(' ')
        cpu_usage = -1
        mem_usage = -1
        try:
                cpu_usage = float(output[1])
                mem_usage = int(output[2])/1024.0
        except:
                globvar.logger.error('Getting CPU and memory usage error - index out of range!')
                cpu_usage = float(output[0])
```

```
            mem_usage = int(output[1])/1024.0

        if cpu_usage >= 0 and mem_usage >= 0:
            cpus.append(cpu_usage)
            mems.append(mem_usage)

        # batch write
        if len(cpus) == 20:   # 60 seconds
            fh = open(fN, 'a')
            for i in range(len(cpus)):
                fh.write(str(cpus[i]) + ' ' + str(mems[i]) + '\n')
            fh.close()
            cpus = []
            mems = []

def main(argv):
        globvar.init(argv)

        thrd_upong = threading.Thread(name='upong', target=upong.udppong)
        thrd_upong.setDaemon(True)       # the child thread will forcefully exit if parent exits
        thrd_upong.start()

        # collect memory and cpu usage info
        date_time = datetime.today()
        timestr = str(date_time.year) + '-' + str(date_time.month) + '-' + str(date_time.day) + '-' +
str(date_time.hour) + ':' + str(date_time.minute) +    ':' + str(date_time.second)
        fN = './cpu_mem/responder_cpu_mem.' + timestr + '.log'
        while(True):
            #monitor(fN)
            time.sleep(3)

if __name__ == "__main__":
        if len(sys.argv) != 4:
                    print "Please input host ip, controller ip and controller port!"
            print "Usage example: python responder.py 100.1.0.1 100.1.0.1 8080"
                    sys.exit(1)
        main(sys.argv)
```

在命令行中运行：

```
$ python controller.py
```

控制器会读取配置，计算探针矩阵同时启动 HTTP 服务器

接下来启动诊断器，运行：

```
$ python diagnoser.py controller_ip controller_port
```

诊断器需要连接到控制器，来获取探针矩阵。

下一步，在所有服务器上启动反应器:

```
$ python responder.py controller_ip controller_port
```

最后，在所有发射探针的服务器上运行发射器：

```
$ python pinger.py host_ip controller_ip controller_port
```

六、结果分析与结论

Detector 是一个实时的，低负载的，高效的大规模数据中心网络监测系统。其核心是一个由可扩展的贪婪路径选择算法通过最小化探测负载方法精密设计的探针矩阵。同样还有一个有效的基于丢包模式的故障定位算法。分析试验台设备和大规模模拟得到，detector 是高扩展性，特别是其同时保证了低负载，而且能够实时高准确率定位故障的大规模数据中心网络监测系统

七、实验心得体会及建议

通过数据中心技术本门课程的实验，我学习到了大规模数据中心网络的组成和原理，以及大规模数据中心网络中可能产生的各种故障、问题，以及故障和问题产生的原因、可能造成的影响等等，同时了解到了许多对于大规模数据中心网络的前沿科学知识，丰富了自己的知识广度。通过实验，我学习了一些编程方法，特别是以前接触比较少但特别实用的的 python 语言。实验中遇到了一些问题，通过与同学交流和上网查阅相关资料以及学习相关案例的方法，使我的学习能力以及解决问题的能力得到了提升。在之后的学习研究中，及时掌握领域前沿知识，深入挖掘、钻研问题，是进一步学习的必不可少的技能和需要继续坚持的品质。

特别要感谢教授本门课程的施老师，他严谨的治学态度和多样化的教学方法大大提高了我们的学习兴趣以及学习效率，他一针见血的指导更是让我们在学习研究中少走了许多弯路。