CC4303-1 Redes
Profesor: José M. Piquér
Auxiliar: David Miranda
Estudiante: Andrés Calderón Guardia

Control 1

Redes

P1. sockets

P1.1.

Las empresas suelen medir el ancho de banda en megabits por segundo (Mbps)[1], y también les debe interesar que la transferencia de datos pueda realizarse a través de múltiples clientes en simultáneo.

Por ello, como propuesta de prueba de estrés se tendría una cantidad considerable de clientes que enviasen un archivo pequeño cada uno en paralelo, medir el tiempo total que demora la transferencia de datos usando el comando `time`, escribir toda la información devuelta por el servidor en un mismo archivo, para finalmente calcular los Mbps con esta información, y como medida adicional variaría el tamaño del buffer en cada caso con el propósito de elegir un valor conveniente tal que se obtenga un mejor ancho de banda que al probar solamente uno.

De esta forma, el servidor que finalmente se elegiría sería el que está implementado con *select*, debido a su alta eficiencia para transmitir información en paralelo con pocos datos por cliente.

P1.2.

Cada servidor tiene una implementación distinta, para el caso de `server_echo2.py` está hecho con *fork*, el cual al paralelizar mediante procesos pesados, conviene para el caso en el que hay pocos clientes conectados a la vez, y estos están transfiriendo una cantidad de datos considerables.

Pasando a `server_echo5.py`, este funciona mediante *select*, el cual es muy eficiente pero se destaca principalmente al realizar muchos procesos pequeños en paralelo.

Por último, `server_echo4.py` utiliza *threading* de Python, el cual funciona mejor como punto intermedio de estos dos casos, maneja mejor varios procesos en paralelo en comparación al *fork* mientras no manejen una gran cantidad de datos, y también es capaz de gestionar una cantidad considerable de clientes a la vez pero no tantos como lo lograría *select* si es que se envían pocos datos.

P1.3.

Para este caso, dejaría ejecutando el servidor a elección en algún puerto en una sucursal central para la empresa, de modo que siempre esté recibiendo un flujo constante de datos de clientes de las distintas sucursales, y con ello utilizar una herramienta externa que monitoree este canal, de forma que sea capaz de detectar estas fallas o congestiones en el sistema para las distintas dependencias.

P1.4.

Dado que se requiere medir el ancho de banda, solo nos interesa obtener la máxima transferencia de datos posible de la red, y dado que los paquetes UDP son más eficientes en su envío, el valor medido usando estos será más cercano al real generando una mejor medición, pese a que estos puedan perder paquetes serían pocos en comparación a la cantidad ganada mediante UDP.

P1.5.

Esta situación, además de depender del ancho de banda, también necesita tener en consideración las limitaciones del propio servidor, de modo que hay que analizar las capacidades de la red y el servidor[2].

[1] https://www.solarwinds.com/resources/it-glossary/network-bandwidth
[2] https://kirbtech.com/what-is-network-server-capacity-planning/

Al aumentar la cantidad de clientes esto supone una carga adicional para el server en variados aspectos tales como el ancho de banda, el uso de CPU, memoria y almacenamiento, entre otros, los cuales en combinación son un factor limitante que afectarían las mediciones, de forma que habría que ir aumentando gradualmente los clientes conectados a la vez para finalmente hallar un valor razonable el cual el servidor pueda manejar.

P1.6.

- server_echo2.py

```
import os, signal, sys, jsockets

MAXPROCS = 50
chld_cnt = 0

def childdeath(signum, frame):
    global chld_cnt
    os.waitpid(-1, os.WNOHANG)
    chld_cnt -= 1

def server(conn):
    print('Cliente conectado', file=sys.stderr)
    while True:
        try:
            data = conn.recv(1024*1024)
            if not data: break
        except:
            print('socket exception', file=sys.stderr)
            sys.exit(1)

        try:
            conn.send(data)
        except:
            print('Cliente desconectado', file=sys.stderr)
            sys.exit(1)

    conn.close()
    print('Cliente desconectado', file=sys.stderr)
    sys.exit(0)

signal.signal(signal.SIGCHLD, childdeath)
s = jsockets.socket_tcp_bind(1818)
if s is None:
    print('could not open socket')
    sys.exit(1)

while True:
    conn, addr = s.accept()
    print(f'Cliente {chld_cnt} conectado')
    if chld_cnt >= MAXPROCS:
        conn.close()
        continue
    pid = os.fork()
    if pid == 0:
        s.close()
        server(conn)
        sys.exit(0)
    else:
        chld_cnt += 1
        conn.close()
```

- server_echo4.py

```
import os, signal, sys, threading, jsockets

MAXPROCS = 50
semaphore = threading.Semaphore(MAXPROCS)

class ClientThread(threading.Thread):
    def __init__(self, addr, s):
        threading.Thread.__init__(self)
        self.sock = s

    def run(self):
        print('Cliente Conectado', file=sys.stderr)
        while True:
            try:
                data = self.sock.recv(1024*1024)
                if not data: break
            except:
                print('socket exception', file=sys.stderr)
                return
            try:
                self.sock.send(data)
            except:
                print('socket exception', file=sys.stderr)
                return

        self.sock.close()
        print('Cliente desconectado', file=sys.stderr)
        semaphore.release()

# Main
s = jsockets.socket_tcp_bind(1819)
if s is None:
    print('could not open socket', file=sys.stderr)
    sys.exit(1)

while True:
    semaphore.acquire()
    conn, addr = s.accept()
    newthread = ClientThread(addr, conn)
    newthread.start()
```

- server_echo5.py

```
import select, os, signal, sys, jsockets

MAXPROCS = 5
active_clients = 0

Sock = jsockets.socket_tcp_bind(1820)
if Sock is None:
    print('could not open socket')
    sys.exit(1)
# Sock.setblocking(0) # revisar si es necesario

inputs = [Sock]

while inputs:
    readable,writable,exceptional = select.select(inputs,[],inputs)
    for s in exceptional: # cerramos sockets con error
        print('Cliente desconectado (error)', file=sys.stderr)
        inputs.remove(s)
        try:
            s.close()
        except:
            pass

    for s in readable:
        if s is Sock:
            if active_clients < MAXPROCS:
                conn, addr = s.accept()
                print(f'Cliente conectado desde {addr}', file=sys.stderr)
                inputs.append(conn)
                active_clients += 1
            else:
                conn, addr = s.accept()
                conn.close()
        else: # leo datos del socket
            try:
                data = s.recv(1024*1024)
            except:
                data = None

            if not data: # EOF, cliente se desconectó
                print('Cliente desconectado', file=sys.stderr)
                inputs.remove(s)
                s.close()
                active_clients -= 1
            else: # Hago eco como debe ser
                try:
                    s.send(data)
                except:
                    print('Cliente desconectado (error)', file=sys.stderr)
                    inputs.remove(s)
                    active_clients -= 1
```

P2. DNS

P2.1. Algoritmo Resolver

P2.1.1.

Se utiliza una implementación inspirada en *weighted round robin* (WRR)[3] para el pseudocódigo, asumiendo que en la lista se proporcionan los pesos correspondientes, indicando que un mayor valor significa una mejor prioridad (los valores de la lista serían de la forma [ns, weight]):

```
1 index ← 0
2 count ← 0
3 function GetNameServer(ns_list)
4 |  global index, count
5 |  if count == 0 do
6 |  |  index ← (index + 1) % length(ns_list)
7 |  |  count ← ns_list[index][1]
8 |  count ← count - 1
9 |  return ns_list[index][0]
```

P2.1.2.

Suponiendo que no haya demasiados NS para consultar, sería una buena opción paralelizar el trabajo para elegir aquel que de la respuesta más rápida, en caso opuesto, generaría un sobrecosto, e inclusive si la cantidad de preguntas realizadas a los NS fuese exuberante, pero por lo general estos casos no se darían ya que se necesitaría una cantidad considerable de NS y/o *resolvers* para que supere el beneficio.

Sin considerar estos casos más extremistas, esto funcionaría dado que el NS que se demore menos en responder será el entregado, de modo que si alguno está congestionado o con fallas entonces responderá otro antes, y por lo tanto, la carga de trabajo se equilibra automáticamente como bono a entregar la respuesta más rápida.

P2.2. DNS obsoletos respondiendo

Sí, esto es grave, pues en primera instancia, genera una discrepancia entre los registros de en este caso ClouDNS y NIC Chile, provocando que los servidores sigan redirigiendo a estos dominios inexistentes, además de que los usuarios que consulten por este dominio en ClouDNS ya que recibirían una resolución inválida o desactualizada.

En la práctica, este problema puede ser bastante peligroso ya que es posible realizar ataques como *DNS spoofing*[4] ya que un atacante podría aprovechar esta inconsistencia para registrar el respectivo dominio con el fin de redirigir el tráfico a sitios maliciosos e incluso recolectar información personal mediante *phishing*.

P2.3. Contienda de Competencia

P2.3.1.

Si todo el sistema DNS se mantuviera centralizado en un solo servidor, sería muy difícil gestionar y escalar la operación de todos los dominios. Por ello, es ideal realizar una división entre la zona padre e hijo para que los dominios o subdominios puedan ser manejados de forma independiente, mejorando la administración y escalabilidad.

Además, también mejora la eficiencia dado que la zona padre al poseer su lista con los servidores del dominio evita tener que recorrer por completo el árbol de búsqueda para hallar el servidor.

[3] https://en.wikipedia.org/wiki/Weighted_round_robin
[4] https://en.wikipedia.org/wiki/DNS_spoofing

P2.3.2.
En este caso, quien manda es la zona del dominio, dado que esta poseerá la información más actualizada respecto a la lista de servidores del dominio, no como en el caso de la zona padre, la cual pese a ser manejada por el DNS, depende del servidor primario notificar de los cambios correspondientes a la zona padre para que se actualice en el *resolver*.

P2.3.3.
Si no hay ningún servidor en común entre la zona padre y la zona primaria, la resolución del dominio fallaría en la mayoría de los casos, ya que los clientes no tendrían una ruta clara hacia los servidores correctos para obtener la información actualizada del dominio si se inicia la consulta desde el padre o una zona superior.

P3. Proxy https

La forma en que funciona CONNECT[5] es que utiliza el primer mensaje para conectarse al servidor, y tras esto, actúa de forma que para el resto de la comunicación simplemente transmite la información codificada entre el cliente y el server, dado que este no la interpreta al usuario.

En cambio, *mitmproxy*[6] además de poder realizar lo mismo que hace CONNECT también es capaz de desencriptar los datos obtenidos a través de este canal, de forma que tanto el cliente como el servidor son capaces de ver, grabar y alterar la información.

Por esta razón, en consideración a la actividad evaluada 2, pese a que ambos métodos sirven como proxies para sitios HTTPS, CONNECT no sería de utilidad ya que en esta se requería llevar un registro del tráfico recibido mediante el *proxy*, lo cual no es capaz de realizar, ya que este no desencripta la información recibida, a diferencia de *mitmproxy* que si puede anotar y analizar esta información.

[5] https://en.wikipedia.org/wiki/HTTP_tunnel
[6] https://mitmproxy.org