Using Sshuttle as a service

Mike R Oct 23, 2019,·6 min read

We use Sshuttle every day to route connectivity across our networks, its a great tool thats easy to spin up and configure, and acts as a lightweight SSH-encrypted VPN (without all the hassles and headaches of IPSEC)

If you're unfamiliar with sshuttle, this is a good article describing its features

We use it so much that I started using it a service to make it easier to start, stop and restart my tunnels, and I am managing my tunnels via Saltstack configuration

This articles shows how to set it up as a service on Centos 7

All the following steps can be done automatically with a **Saltstack formula**

(If youre using Puppet or Ansible, the steps are pretty much the same, just tailor it to your specific tool)

All the scripts below are also hosted here: **Github**

1 — Service Account

On the server initiating sshuttle (client), create a dedicated Sshuttle service account and create an SSH folder

```
root@client>
groupadd sshuttle
useradd -d /home/sshuttle -g sshuttle sshuttle
mkdir /home/sshuttle/.ssh
chown -R sshuttle:sshuttle /home/sshuttle
chmod 700 /home/sshuttle/.ssh
```

generate a secure SSH key

```
root@client>
ssh-keygen -o -a 100 -t ed25519 -N "" -C "sshuttle_key" -f
/home/sshuttle/.ssh/id_ed25519
```

This will generate an ed25519 key pair

distribute the public key to whatever host you want to connect to (good practice is to create this service account on each host, and add this key to this sshuttle account's Authorized_keys file)

try to connect to the target server as sshuttle user to test basic SSH connectivity

```
root@client> su sshuttle
sshuttle@client> ssh targetServer
```

if you can SSH to the target, move on to next step

2 — Sudo access

Sshuttle client needs sudo access to modify your firewall (on client only, not on the target server)

add the following to “/etc/sudoers.d/sshuttle”, make sure theres an empty line before and after the sudo line

```
sshuttle ALL=(root) NOPASSWD: /usr/bin/python /usr/share/sshuttle/main.py
/usr/bin/python --firewall 12*** 0
```

this allows non-root users (like our service account) to launch Ssshutle and modify the firewall with ports 12xxx

3 — Install package

install Sshuttle on your client server

```
root@client> yum install sshuttle
```

4 — Service scripts

add 2 service scripts,

1st is a SystemD script that controls Sshuttle (for Init.d systems, see this Gist for sample init.d script)
2nd is a Python script that reads in your sshuttle config file

```
root@client> vi /etc/systemd/system/sshuttle.service[Unit]
Description=sshuttle service
After=network.target[Service]
User=sshuttle
Restart=always
Type=forking
WorkingDirectory=/etc/sshuttle
ExecStart=/etc/sshuttle/sshuttle.py start
ExecStop=/etc/sshuttle/sshuttle.py stop[Install]
WantedBy=multi-user.target
```

reload systemd

```
systemctl daemon-reload
```

Now for the 2nd script — this pythons script reads data from your config.json file and starts, stops, restarts the actual Sshuttle binary

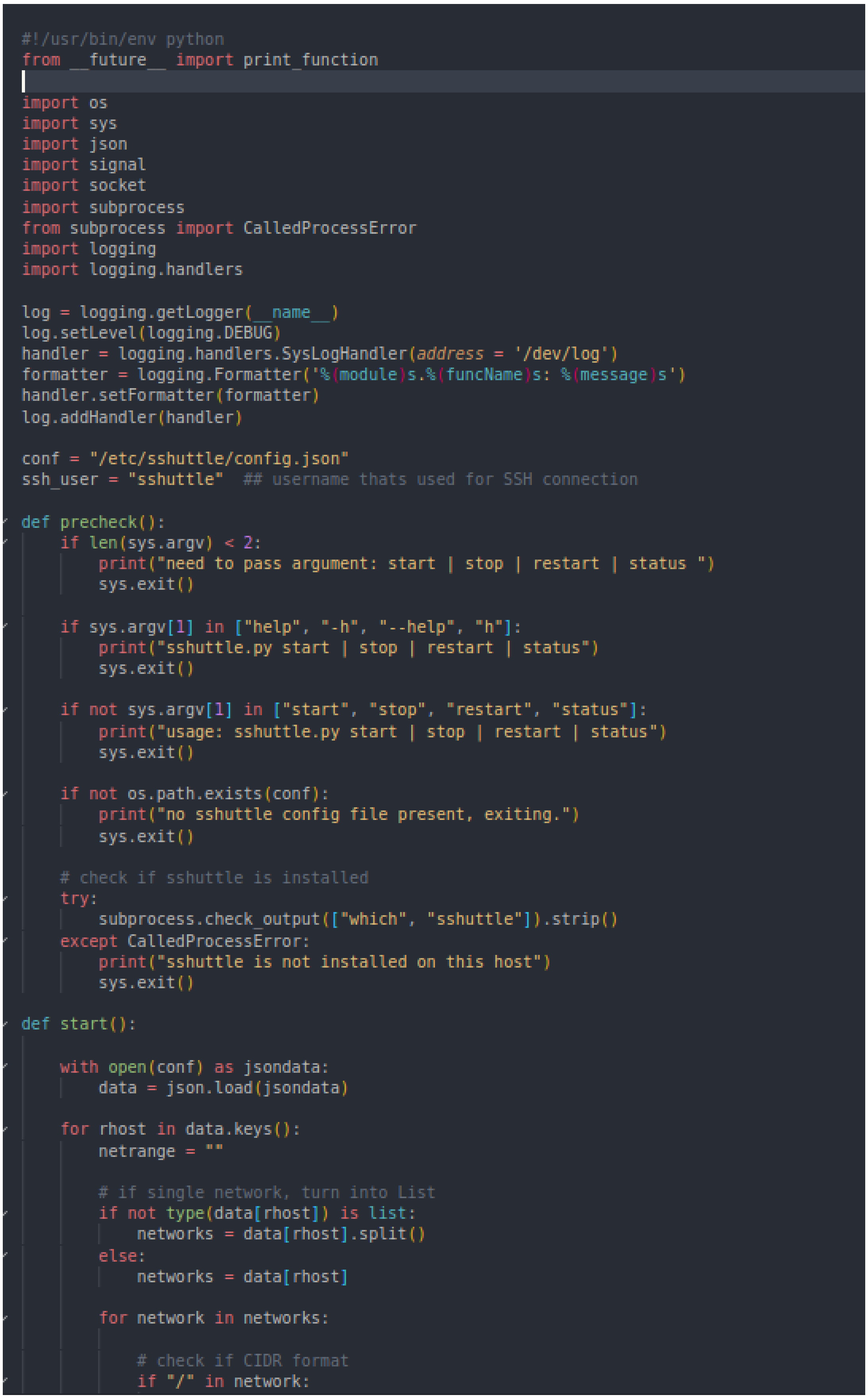
create an /etc/ directory for sshuttle

```
mkdir /etc/sshuttle
chown sshuttle:sshuttle /etc/sshuttle
```

Su as “sshuttle” user and add the python start script, this will read in your Config file (json) and start/stop the tunnel

```
sshuttle@client>  vi /etc/sshuttle/sshuttle.py
```

Save this python script (its on **Github Gist**) as/etc/sshuttle/sshuttle.py



```python
#!/usr/bin/env python
from __future__ import print_function

import os
import sys
import json
import signal
import socket
import subprocess
from subprocess import CalledProcessError
import logging
import logging.handlers

log = logging.getLogger(__name__)
log.setLevel(logging.DEBUG)
handler = logging.handlers.SysLogHandler(address = '/dev/log')
formatter = logging.Formatter('%(module)s.%(funcName)s: %(message)s')
handler.setFormatter(formatter)
log.addHandler(handler)

conf = "/etc/sshuttle/config.json"
ssh_user = "sshuttle"  ## username thats used for SSH connection

def precheck():
    if len(sys.argv) < 2:
        print("need to pass argument: start | stop | restart | status ")
        sys.exit()

    if sys.argv[1] in ["help", "-h", "--help", "h"]:
        print("sshuttle.py start | stop | restart | status")
        sys.exit()

    if not sys.argv[1] in ["start", "stop", "restart", "status"]:
        print("usage: sshuttle.py start | stop | restart | status")
        sys.exit()

    if not os.path.exists(conf):
        print("no sshuttle config file present, exiting.")
        sys.exit()

    # check if sshuttle is installed
    try:
        subprocess.check_output(["which", "sshuttle"]).strip()
    except CalledProcessError:
        print("sshuttle is not installed on this host")
        sys.exit()

def start():

    with open(conf) as jsondata:
        data = json.load(jsondata)

    for rhost in data.keys():
        netrange = ""

        # if single network, turn into List
        if not type(data[rhost]) is list:
            networks = data[rhost].split()
        else:
            networks = data[rhost]

        for network in networks:

            # check if CIDR format
            if "/" in network:
```

This reads in a json config file, parses each hostname or IP and creates a tunnel. This script knows for example if you want to proxy a single IP connection or an entire CIDR subnet

```
# make script executable
chmod +x /etc/sshuttle/sshuttle.py
```

You can now start, stop and restart Sshuttle service using systemd

```
systemctl status sshuttle
systemctl start sshutle
systemctl stop sshuttle
```

5 — Config File

finally, you add a config file to tell sshuttle where you want to connect to and what networks you want to route via the target hop server, you can add multiple hop servers and multiple networks that you want to route via these hop proxies,

Add a new **config.json**

```
sshuttle@client> vi /etc/sshuttle/config.json{
  "HopServerA": [
    "12.182.293.180/32",
    "129.33.78.18/32",
    "129.13.280.0/24",
    "sftp.somehost.com"
  ],
  "HopServerB": [
    "11.38.26.0/24"
  ]
}
```

Start the tunnel

```
systemctl restart sshuttle
```

If you are using Saltstack, you can manage network tunnel configs for your hosts by passing data pillars like this,

```
cat /srv/salt/pillar/servers/nycweb01.slssshuttle:
  HopServerA:
    - 12.182.293.180/32  # customer A
    - 129.33.78.18/32    # customer B
    - 129.13.280.0/24   # customer C
    - sftp.somehost.com  # ftp customer DHopServerB:
    - 11.38.26.0/24  # customer F
```

the Saltstack init.sls formula (in the link above) will generate a **config.json** file based on this Pillar

This makes it easy to manage your Sshuttle tunnels via managed configuration code.

—

Update #1 — Keep Alive

I also updated this part of the sshuttle.py script, with some additional settings (based on my work performance testing)

```
rpath = "-r {0}@{1} {2} -l listen '0.0.0.0' --ssh-cmd 'ssh -o 
ServerAliveInterval=60' --no-latency-control".format(ssh_user, rhost, netrange)
```

This will allow your shuttle to listen on all interfaces so you use this host as a hop. If you want to keep your sshuttle session private, change the 0.0.0.0 to 127.0.0.1

and this additional part

```
--ssh-cmd 'ssh -o ServerAliveInterval=60' --no-latency-control'
```

makes sure your SSH heartbeat is resent every 60 seconds so your SSH connection doesnt go stale, and that sshuttle doesn’t throttle your bandwidth on high speed networks.

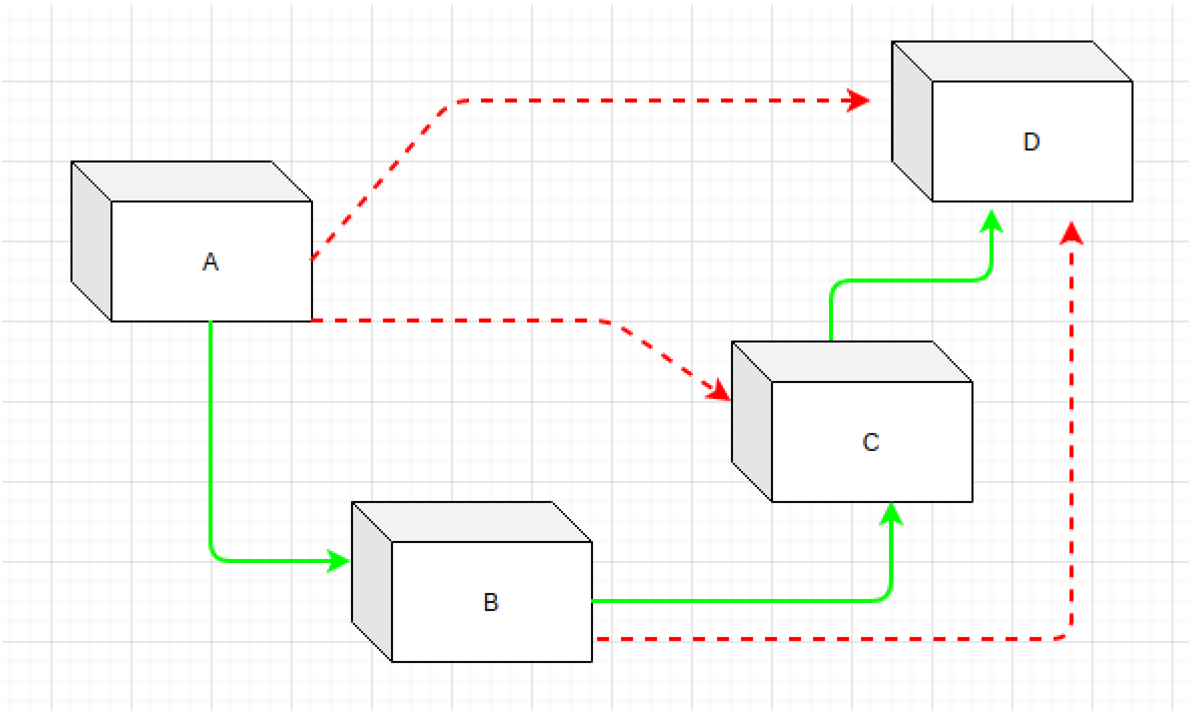
—

Update #2 — Proxying via 2 or more hops

if you need to use sshuttle to hop over 2 or more hops to get to a destination, for example

server A needs to get to server D
server A can connect to server B but not C,D
server B can connect to server C but not D
server C can connect to D

your hops will look like this:
A > B > C > D



Picture 1: A needs to get to D, via B and C

the regular sshuttle configuration will allow you to hop A > B > C only

to get to D, you need to configure your sshuttle to use C as a hop to D, only after the 1st connection is established between A and B

Configure your config.json on server A, adding colon-separated steps to each connection

route all connections to C via B (step 0), and all connections to D via C (step 1)

Shuttle will read the step sequence and initiate the Step 0 connection 1st, once its established, it will start Step 1 connection 2nd (since you need Step 0 to be up before you can run Step 1)

Adding these steps ensures your connections come up in proper sequence.

```
# config.json on server A{
  "0:serverB": [
    "serverC hostname or IP",
 ],
  "1:serverC": [
    "serverD hostname or IP"
 ]
}
```

restart sshuttle service on serverA

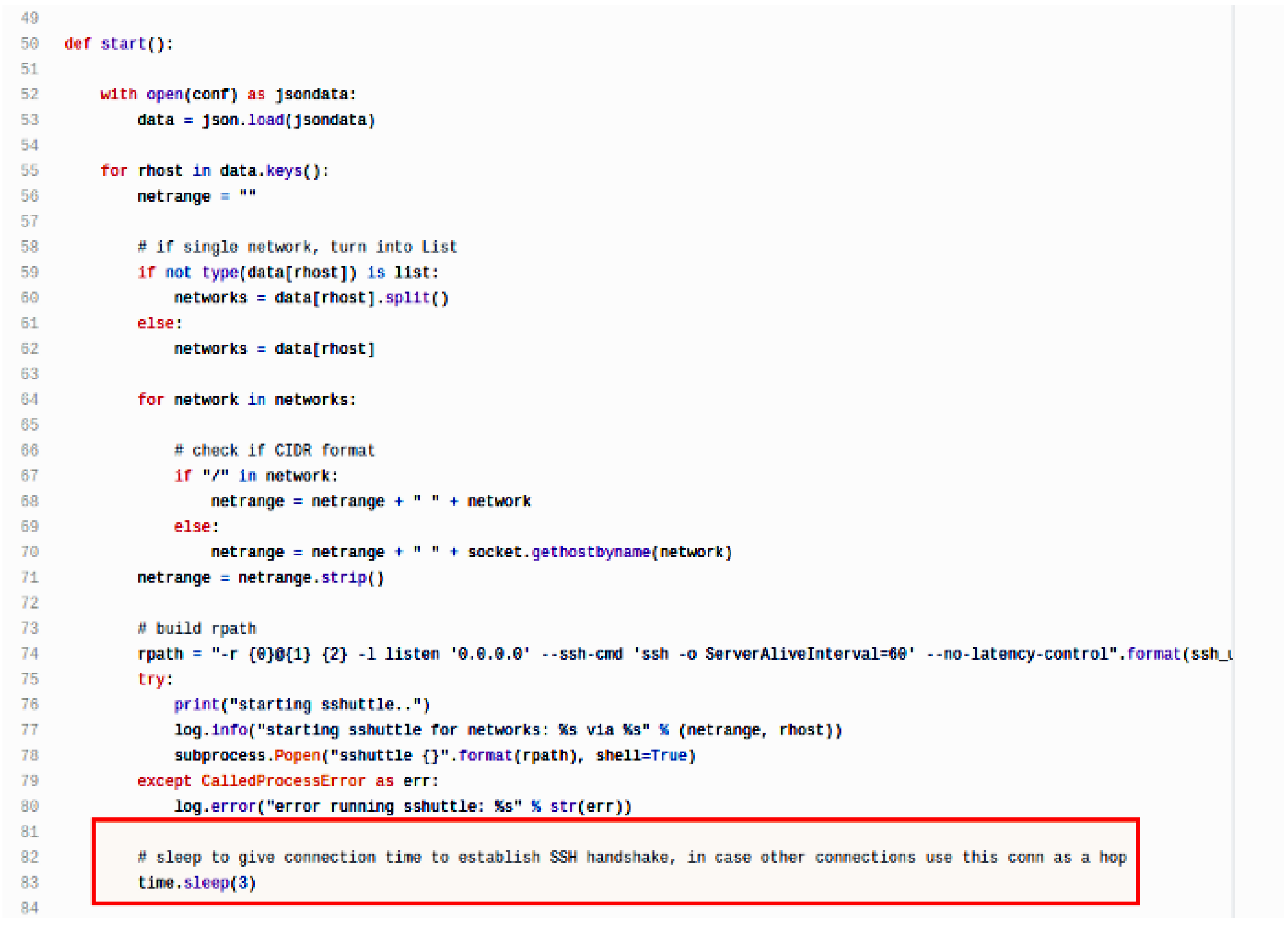
on server B, configure your config.json to route all D-bound routes over C (here, you dont need to add steps since its a single connection)

```
# config.json on server B
{
  "serverC": [
    "serverD hostname or IP"
  ]
}
```

restart sshuttle on server B

sshuttle iterates your config.json on serverA, and looks at the 1st connection (B>C)

it will then sleep for 3 seconds to allow this handshake to establish



```
def start():

    with open(conf) as jsondata:
        data = json.load(jsondata)

    for rhost in data.keys():
        netrange = ""

        # if single network, turn into List
        if not type(data[rhost]) is list:
            networks = data[rhost].split()
        else:
            networks = data[rhost]

        for network in networks:

            # check if CIDR format
            if "/" in network:
                netrange = netrange + " " + network
            else:
                netrange = netrange + " " + socket.gethostbyname(network)
        netrange = netrange.strip()

        # build rpath
        rpath = "-r {0}@{1} {2} -l listen '0.0.0.0' --ssh-cmd 'ssh -o ServerAliveInterval=60' --no-latency-control".format(ssh_u
        try:
            print("starting sshuttle..")
            log.info("starting sshuttle for networks: %s via %s" % (netrange, rhost))
            subprocess.Popen("sshuttle {}".format(rpath), shell=True)
        except CalledProcessError as err:
            log.error("error running sshuttle: %s" % str(err))

        # sleep to give connection time to establish SSH handshake, in case other connections use this conn as a hop
        time.sleep(3)
```

Once the 1st connection is up, it will attempt to connect C > D via this 1st connection (this happens on serverB)

Also, by default sshuttle listens on 127.0.0.1 for any incoming requests, so on **server B,** make sure you configured the **sshuttle.py** to listen on all interfaces, not just `127.0.0.1` , this will allow B to proxy any incoming requests from external hosts, not just requests coming from itself.

if using Saltstack:

```
rpath = "-r {0}@{1} {2} -l {{ salt['pillar.get']('sshuttle:listen', '127.0.0.1')
}} --ssh-cmd 'ssh -o ServerAliveInterval=60' --no-latency-
control".format(ssh_user, rhost, netrange)
```

and pass the Pillar data like this (if no pillar “listen” data is provided, it defaults to listening on 127.0.0.1

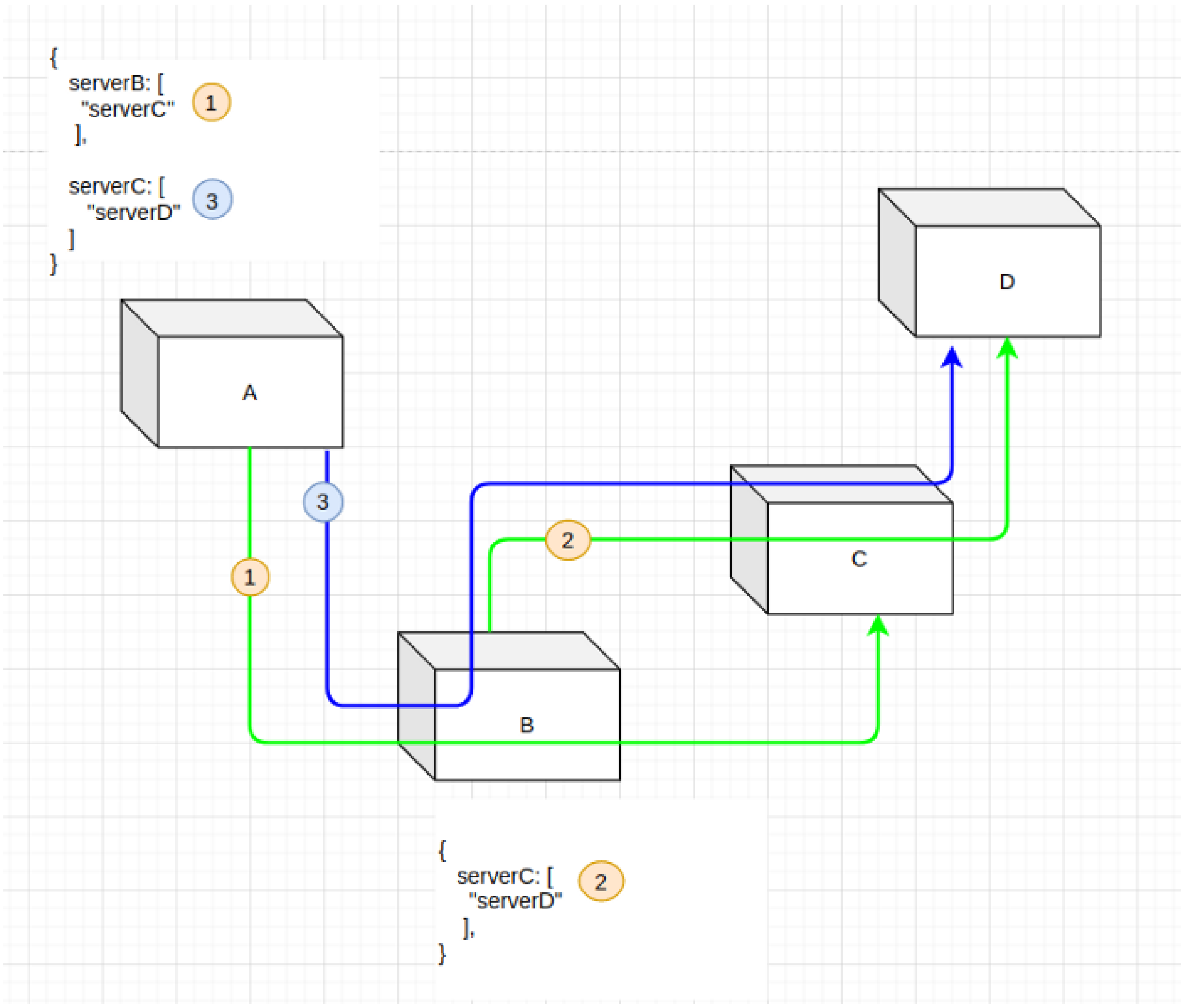
```
## Server B pillar
sshuttle:
  listen: 0.0.0.0
  relays:
    serverC:
      - serverD ip or hostname
```

if you are not using Saltstack, just update /etc/sshuttle/sshuttle.py directly on **server B** to listen on all interfaces

```
rpath = "-r {0}@{1} {2} -l 0.0.0.0 --no-latency-control".format(ssh_user, rhost,
netrange)
```

now when you initiate a connection from A, it will proxy like this:
A > B > C > D

Your final packet flow will look like this


```
{
  serverB: [
    "serverC"
  ],

  serverC: [
    "serverD"
  ]
}
```

```
{
  serverC: [
    "serverD"
  ],
}
```

```
====================  E n d e   i m   G e l ä n d e  ====================
```