Cheap Portable k3s

Sachin Iyer

January 27, 2023

1 Overview

This is the description of how I created my k3s cluster with tailscale and old thinkpads. I believe that the architecture is somewhat unique, and fits the use case of broke college kid trying to deal with no public ips and moving all the time. These docs are currently a work in progress, and contain a lot of my musings on the trials and tribulations taken when building this cluster.

1.1 Cheap

I needed this cluster to be somewhat cheap for two reasons. First, I don't have any high sla requirements, nor do I really care too much about performance (gives me more time to stop and touch grass). As such, it is better for me to try and create something that optimizes for price.

1.2 Portable

In my youth, I tend to move around quite a bit, and it is an absolute pain to set up port forwarding everywhere that I go. I also am not guaranteed access to a public IP wherever I am going. Therefore it is necessary that I am able to move my cluster with ease.

1.3 Secure

The previous version of this cluster used to be accessible by ssh with a 5 letter alpha password. This is just not smart. I wanted to make this cluster slightly more secure and control the traffic that hits the cluster better.

1.4 k3s

I really wanted to make a kubernetes cluster not just to learn, but because it allows me to deploy all of my apps with ease. I also get to keep my data on my machines and overall just have a testing ground for all of my stuff that I do. If I want something to go up, I am not at the mercy of my current employers cloud service.

2 Hardware

There are two thinkpads and one fanless computer in my 3 node k3s cluster. I also have an ec2 instance that acts as my "entrance node"' and also headscale control server. I don't include this in the cluster, because I want an odd number of nodes.

2.1 Compute

I have a very powerful 12 virtualized cores in this cluster. It may not seem like much, but I paid $250 in total.

Thinkpads The cheapest way to get cores is to buy used thinkpads (to which I am preferential anyway). The two secondary nodes are thinkpad T410s. These thinkpads have been semi-reliable for the last year during the formation of this cluster. You can also look into removing the Intel ME as well as installing coreboot if you have a computer that is old enough (the T410s are). The computer will also boot if you short two of the pins in the eDP ribbon connector (which means you can remove the display).

Fanless Computer from China The master node is a cheap fanless computer. I decided to go with a new fanless computer for the master node to increase reliability a bit more. It is also because there were no T410s on new york craigslist when I was expanding to my last node

Why no raspberry pi I did not use raspberry pis because a lot of my applications do not play nice with arm, and they are extremely expensive right now. In the previous iteration of this cluster I ran a multiarch setup

with x86 and arm, and I ended up running into a lot of weird problems. I decided to avoid the headache and just stick to x86.

2.2 Storage

I don't want to deal with hdds - I removed all of them. Spinning disks are really quite annoying. Each node has an internal boot drive (128 or 256 gb) and then an external 256gb ssd. In total I have 768gb of storage available.

2.3 Physical Networking

TP Link Access Point I use a TP Link Router (in AP mode) to both extend my wifi network in my apartment and also connect the nodes. I don't worry too much about networking speeds, but everything is theoretically around gig-speed.

TP Link Switch I also have a small switch that I use for the rest of the machines and some raspberry pis that I have.

3 Networking

This is the heart of this project. The main motivation for remaking this cluster was to create something that I could move from place to place with me very easily.

3.1 Tailscale

3.1.1 Reasoning for Tailscale

What is Tailscale Wireguard is a way to establish quick p2p encrypted tunnels at a kernel level. Tailscale handles the key orchestration for wireguard. The end result of this is a p2p connection with every other machine in the "Tailnet". Using some fancy port opening stuff, you can also expand your wireguard tunnel through NATs and essentially communicate with every other machine no matter where you are.

Why Tailscale When all the nodes of the cluster and machines used for controlling the cluster are in a "Tailnet" together, you essentially have a private network where not only can you access internal service (one of the advertised uses for tailscale), but also machines can talk to each other without worrying about where they are. You can also manage individual nodes without needing to be in the same network. There is no more tricky, finicky, insecure port forwarding.

What's Headscale Headscale is an awesome open source project that allows you to self deploy a server with many of the same capabilities as the proprietary Tailscale server. This means that I don't have to worry about a device limit and can keep all keys on my own machines. Since the Tailscale clients are open source, I actually can just use the Tailscale clients to connect to my headscale instance (including the android app).

Another cool thing about Tailscale is that I can still use their DERP and relay servers without issues. Although headscale can be deployed with a DERP server, I found this to be very finicky.

3.1.2 How do packets move

Entrance Nodes This is the term that I started using for traffic that comes into the cluster. These nodes need a public IP and basically do an nginx proxy_pass. I use an ec2 instance for this purpose. I may swich back to a machine with port-forwarding if I find a place that I can get a public IP with.

Another way to do this would be to just configure IP tables to pass the packets from the public IP to the tailscale IP. I don't do this however, because I want to validate it as an HTTP request first. I also prefer to configure nginx and route based on domain name instead of blindly forwarding packets.

You can actually visit a couple of non-cluster machines that are connected to the network through this entrance node, such as my laptop (if it is up), Raspberry Pi 1, and Raspberry Pi 2.

Exit Nodes This is the tailscale term for machines that advertise themselves as being able to forward your traffic through them. I actually use a rapsberry pi for this purpose. I also connect that raspberry pi to my preferred VPN, so that anytime I want a VPN connection, I don't have to use

an external application. I can also add a VPN connection very quickly to any node just by configuring it to use the exit node.

TLS Handling TLS is a bit tricky. I prefer to do ssl termination on the cluster or end machine instead of the entrance node itself, This way, I can keep the traffic encrypted while traveling to machine as well as take advantage of cool tools like cert-manager. On nginx, you use a combination of proxy_pass and SNI (Server Name Indication) to get your packets going to the right place. You can also configure Traefik (my ingress of choice) and Nginx (my other ingress of choice) to accept proxy_passed packets. Also depending on the Load Balancer that you are using, you may have to configure that to accept proxy_passed packets as well.

Domain Names Another requirement for this project was to be able to seperate internal and external tools. I do this through whitelisting domains at the entrance nodes. Instead of blindly forwarding all the traffic from *.sachiniyer.com, I instead just forward the tools I want available to the world. Traffic that reaches the entrance nodes with those domains will be allowed into the tailnet.

All traffic with other domains will have to be generated from inside the tailnet. You can make this easier with a quick change to resolv.conf to point traffic from your domain to the magicDNS of the cluster. This results in a clear separation of internal and external tools. You could also configure your dns record to only have the whitelisted domains as well.

3.1.3 Tips and tricks

As a couple of tips, it can be hard to keep the nginx config consistent among all of your entrance nodes, so what I recommend doing is keeping git as a source of truth, and do a cronjob that pulls from a git repo and automatically updates at whatever frequency that you would like. There may be a better way to do this with webhooks. I would also avoid keeping your nodes in restrictive networks, as this means that they use the relay servers as little as possible and you have faster speeds.

3.2 Klipper

This is somewhat standard, but I use klipper (or servicelb) as the bare metal load balancer for this project. This sits on the tailscale magicDNS and works perfectly to distribute traffic accross the nodes. I also have multiple targets in the nginx config so that traffic is somewhat balanced from there as well. If one machine goes down, theoretically traffic should be forwarded to the node that is up and everything should proceed as normal (this has happened once - and to my surprise everything worked as expected).