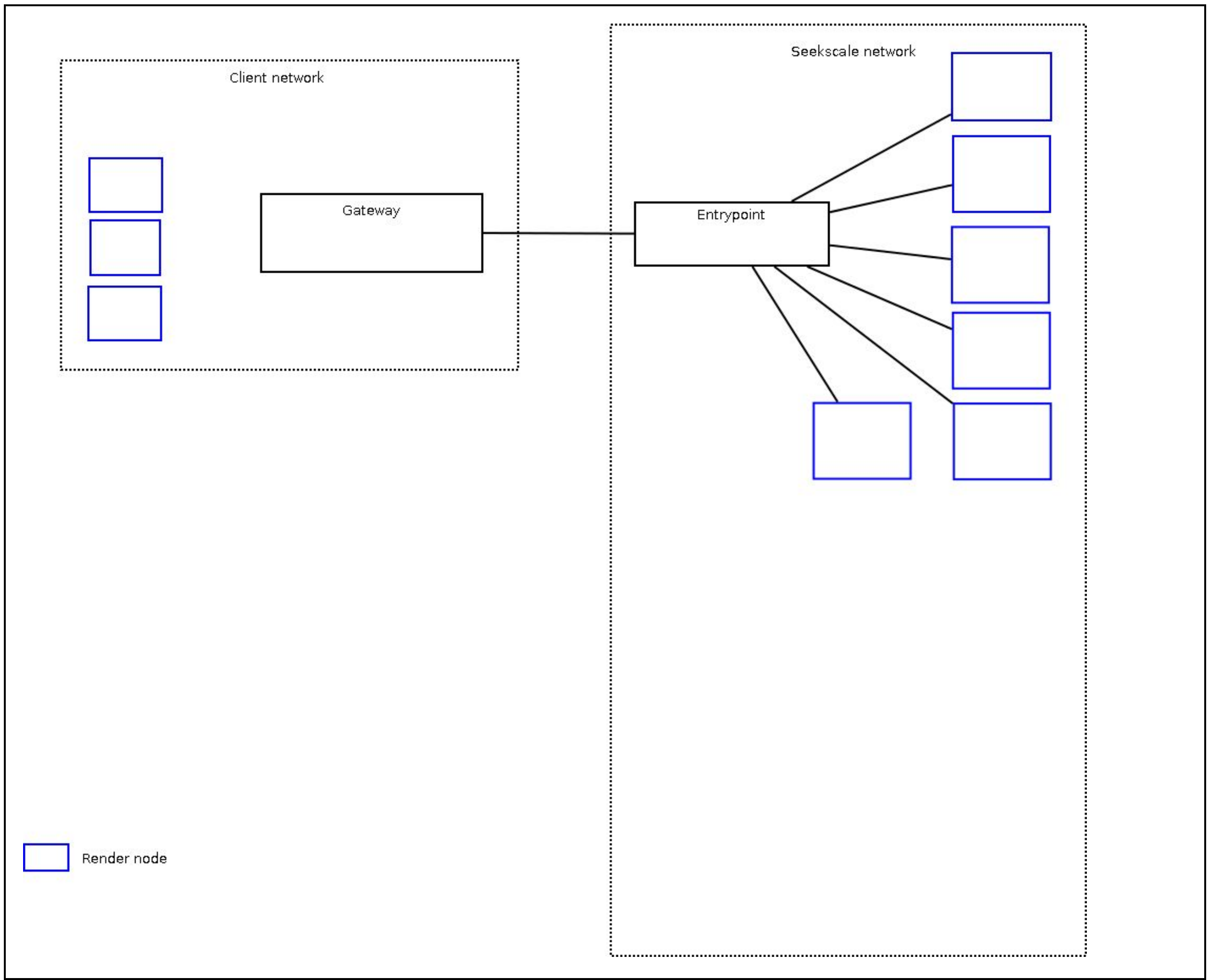
This is an introductory document of the main Seekscale/smbproxy systems, what they do, how they interact with each other, etc.

Goal

The goal of a Seekscale deployment is: given a client studio, with its own internal network with a file server, license servers, and a few render nodes, how do we setup render nodes that are on *our* side (mostly EC2 or OVH servers), and give them access to the client's internal network?

Network topology

The typical network organisation for a Seekscale deployment is shown in figure [Network-topology-1].



[Network-topology-1]

The machines we have are:
* the gateway. It is usually a VM installed on the client's internal network.
* the entrypoint. It is the central server, that will handle all the network connections, host the cache, and monitor everything.
* the render nodes. The machines that we give to the client, that do the work.

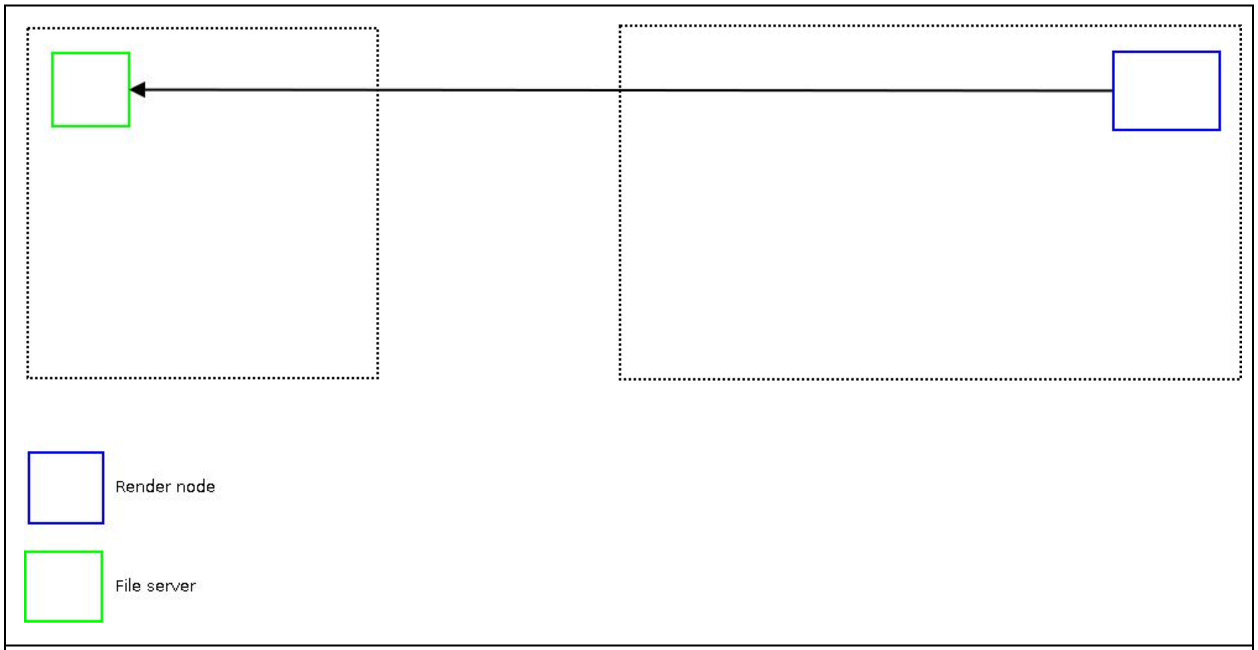
Connection is made by two openvpn networks. For both networks, the server is on the entrypoint.
One network is between the gateway and the entrypoint. It uses the network 10.91.1.0/24. The gateway always has the ip 10.91.1.1, and the entrypoint always has 10.91.1.254
One network is between the entrypoint and the render nodes. It uses the network 10.91.0.0/24. The entrypoint always has the ip 10.91.0.1

Once everyone is connected, we add routes and NAT rules so that our render nodes can connect to the client's network using the usual ips of the client.
To do that, we add:
* routes in the openvpn configurations so that any traffic to anything that looks like a private ip (192.168.*, 10.*) is directed inside the vpn, to the entrypoint, then to the gateway
* on the gateway, setup a NAT so that when the packets reach the client's network, they can come back.

The client will also want to connect to the each render node individually. To do that, there are two methods:
* setup a bi-nat on the gateway. A bi-nat is a nat where each server behind the nat gets its own dedicated ip. Basically, what this means is that to each render node on the Seekscale side, we associate an ip address from the studio's internal network. This is the easiest to setup for us, but it requires that the gateway is running linux. There is no way to do that when the gateway is running Windows 7, and doing it when the gateway is running Windows Server is possible, but quite complicated (and sometimes it doesn't work, for unknown reasons).
* setup routing on the client network, so that their network knows how to route the entire 10.91.0.0/16 network towards us. This is a bit more complicated, because the client has to configure it by themselves (and usually, they don't know how to do it). But it always works.

The smbproxy

Once the network topology step is done, the machines can already talk to each other, and the render nodes on the Seekscale side can already read files from the client's file server. Figure [Client-SMB-1] shows what the connection looks like. But there is still a problem: the connection between the gateway and the entrypoint goes over the internet, plus within an openvpn tunnel, and usual file transfer protocols aren't really optimized for this use case. In practical terms, the max bandwidth we can expect is around 10-15Mb, sometimes even less. Even when are working on 300Mb or higher connections.

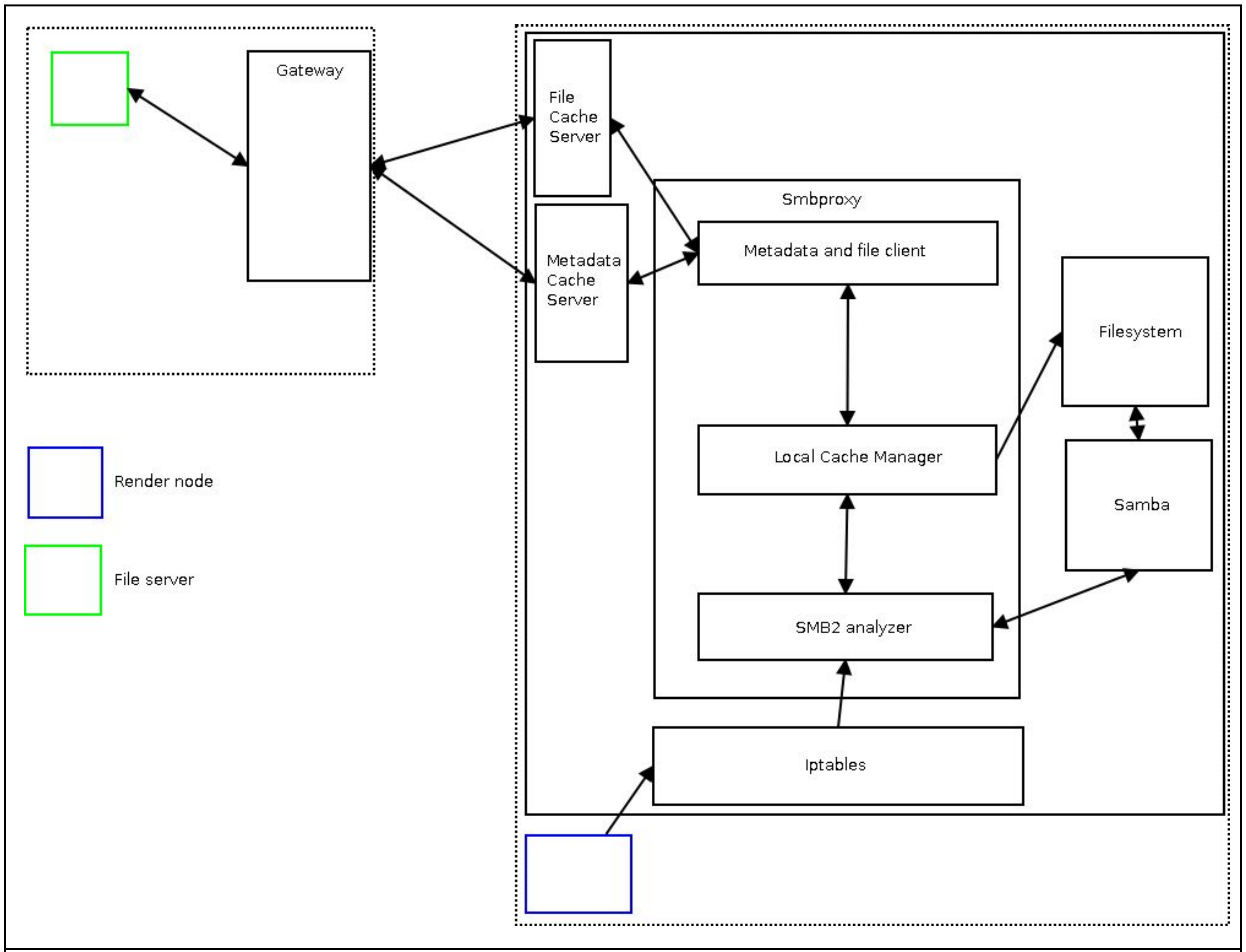


[Client-SMB-1]

The smbproxy is our answer to this problem, for file transfers done using the smb protocol (ie, normal windows shares). The general idea is that, instead of the render node connecting directly to the client's file server, it will connect to a server on the entrypoint, that will intercept the request for the file, ask the gateway to retrieve the file, transfer it from the gateway to the entrypoint using an optimized protocol, then serve the file.

The two main benefits to this system are that:
* we control the way file are transferred between the gateway and the entrypoint, which means we can use any protocol that is adapted to the high latency
* we can keep a cache of the files on the entrypoint, which means that if a render node requests the same file multiple times, or if multiple render nodes request the same file, the file only has to be transferred once from the gateway to the entrypoint

Figure [Client-SMB-2] shows all the parts of the system, and how they interact with each other.



[Client-SMB-2]

The process followed, when a render node tries to open a file, is the following:

1. The render node tries to connect to the ip address on the file server, and to send a "OPEN" SMB2 packet.
2. The connection passes through the entrypoint, which, using iptables rules, will intercept the connection attempt, and redirect it to the "smbproxy" process, listening on the entrypoint itself.
3. The smbproxy process receives the connection. It will use its SMB2 protocol analyzer, to try to figure out what the render node wants. In this case, it will interpret that the render node wants to open a file.
4. The smbproxy now needs to ensure that a copy on the file that the render node asked for is present on the local filesystem. To do that, it will first check if we have already imported the file. Then it will compare the metadata (file size, last modification time) on the seekscale and on the client side.
5. If the local copy is not up to date, the smbproxy will ask the gateway to upload the file to the entrypoint, and more precisely into the File Cache Server. The way the File Cache Server works

is described in a later section, but for simplicity, one can say that it's a HTTP file server that supports upload.
6. If a file has been uploaded to the File Cache Server, we transfer it to the local filesystem.
7. We now know that the file on the local filesystem is up to date with the distant file on the client's file server. The smbproxy will now act as a simple proxy would, and forward the packet to a samba server, that will read the filesystem, and serve the file just as usual.

The main trick is that we are using both a "classical" SMB server (samba) and a SMB proxy in front of it. The reason we do this is implementation simplicity. The smbproxy *never* modifies the network traffic. All it does is modify the file that the samba server can see. This way, we don't have to reimplement a full SMB2-compliant server. We just peek at the traffic, and act on what seems interesting. If a SMB2 packet is too complex, we just let it through and let samba handle it, assuming it's an advanced feature that we don't need to interfer with.

The file transfer protocol
What is the most bandwidth-efficient way to securely transfer data between two servers, with potentially high latency?
The current answer we have is:
* HTTPs with client certificate authentication
* Split the file to upload in small (5MB) chunks. Use multiple parallel connections, to maximize the bandwidth usage.
* Make the HTTPs connection *outside* the VPN link, so that we're not subject to openvpn limits.
* Compute a checksum for each chunk before and after the transfer, to ensure that we catch any transfer error.

The client for this protocol is implemented in renderfarm_commons.cache_client.filecache_client3.CacheClient3. It is based on twisted.
For the upload, it concurrently splits the file in small chunks, computes a checksum, checks if the chunk has already been uploaded, and then uploads the chunk. When all the chunks are done uploading, it writes the list of chunks to redis, so that we can retrieve them later.

The server for this protocol is basically nginx+a tiny flask service+redis to store the metadata. We use a nginx feature called client_body_in_file_only, which makes it that nginx directly writes the upload file to a file, instead of passing it to the flask service with the rest of the request. This improves performance *a lot*, especially with large requests, because nginx is much more performant than python for transferring data. So, in our case, all that the flask service do it check that the file has been uploaded, check its checksum, and then use rename() to move it to its final location.

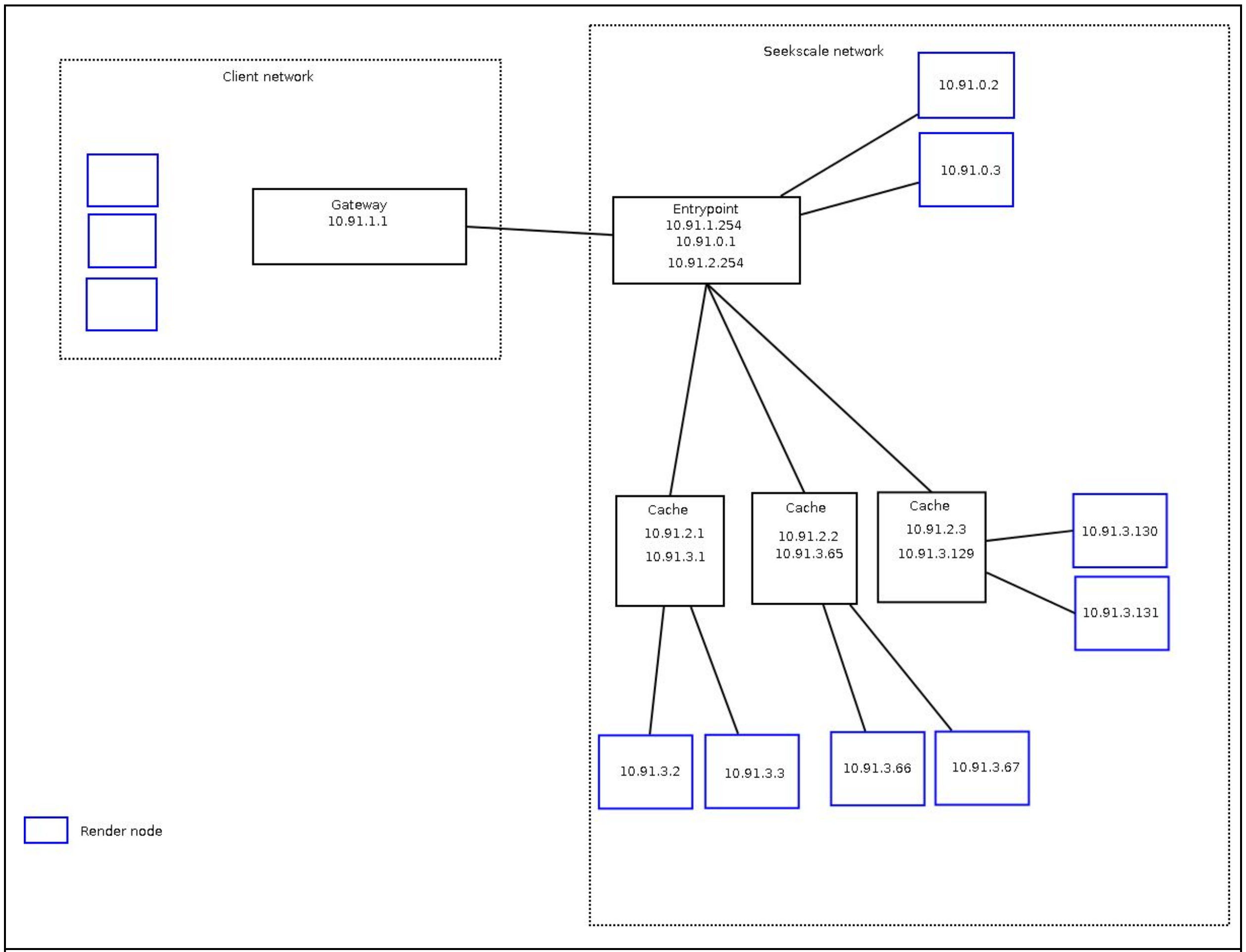
Some notes on this protocol:
* It is mostly an internal implementation detail. It doesn't really have any visible implications. We can switch protocols if we find something more efficient/easier to use. We actually have switched a few times already (we have already tried plain HTTP, SSH, and Openstack Swift).
* It has a bit of overhead. As a consequence, we don't use it for small (<1M) files, which are transferred directly via http within the VPN.

Writing files to the client file server
When a render node wants to write to the client file server, the process is a bit different than for reading.
The SMB2 packets are decoded the same way, but instead of the open() command, we are looking for the close() command, which signals that the client has finished writing to a file.
We then upload the file from the entrypoint filesystem into the File Cache Server, and use a queue (backed by redis) to signal to the gateway that it must download the file.

We use a queue because we need to be able to reliably know what files have been transferred, and what files haven't. In the case of reading, when there is a network cut, or when the entrypoint crashes, it's not a disaster. When the entrypoint restarts, it will, at worst, have to recheck all the files to ensure that they are up to date. On the other hand, in the case of writing, if the entrypoint or the network crashes in the middle of transferring the file, we need to keep a reliable queue, to be able to know what failed to transfer, and to be able to restart the transfers.

Extension 1: Using multiple cache servers
One of the limitations of the topology that is described up to here is that we can't scale up the number of render nodes too high, since they all share the same connection to the entrypoint. The rendernode<->entrypoint connection is an internal one, so it generally has low latency and can go up to 500Mb or 1Gb, but if many render nodes request files at the same time, it can still be a bottleneck.

The solution we use is shown on figure [Network-topology-2]. We add intermediate "cache" machines, and we use a new openvpn network to link all cache machines to the central entrypoint.



[Network-topology-2]

If we go back to figure [Client-SMB-2], we can see how we split the different services between the cache and the entrypoint:
* the smbproxy and the samba server are present on each Cache machine
* the Metadata Cache Server and the File Cache Server stay on the central entrypoint

This way, we now have two levels of cache for the files. One level at the cache machine, which lives on the filesystem, and is accessible directly to samba. One level at the entrypoint machine, which is shared by all the cache machines.
Other than that, the general process doesn't change much.

We similarly have two levels of cache for the metadata, one in the central entrypoint, and one in the memory of each cache server.

As an additional bonus, it turns out that we don't have SMB2 traffic between the entrypoint and the cache servers, so they can be in entirely different datacenters.

Extension 2: Optimizing metadata distribution

In the setup that was described up to here, when a render node requests a file and the cache server doesn't have up to date metadata for it, the cache server asks the entrypoint for it, who then, in turn, will ask the gateway if it doesn't know.

This misses a common case, which is when one of the cache servers doesn't know about a file, but another cache server has requested it very recently. To save the cache server from having to ask the entrypoint in this case, we use a "push" system to distribute the metadata.

This push system is implemented by a redis database on the entrypoint, with a linked slave on each cache server.

In the end, the process to check the metadata of a file, from the point of view of a cache server, is:

1. Check if we have an up-to-date version of the metadata in memory. If we have one, use it
2. Check if we have an up-to-date version of the metadata in the local redis slave. If we have one, use it
3. Ask the entrypoint for an up-to-date version of the metadata. If the entrypoint has one available, it will send it to us.
4. If the entrypoint doesn't have an up-to-date version, it will ask the gateway, and send us the result (plus put the result in the redis master, so that all other slaves get it too).

Note that getting an answer at step 3 should be quite rare. We only get an answer if: the entrypoint has an up-to-date version of the metadata for the file, but it hasn't been written in redis, or the replication hasn't caught up.

The code

Here are where the code for the various components is located:

The smbproxy process lives in the repository git-luna:smbproxy, in package smbproxy4.
The three components that are visible on the diagrams correspond to the three main files of the code:
SMB2 Analyzer => smbproxy4.py
Local Cache Manager => fs_local_cache_client.py
Metadata and File Client => fs_cache.py

The Metadata Cache Server lives in the repository git-luna:smbproxy, in package metadata_proxy. The main file is metadata_proxy.py

The components to run on the gateway live in the repository renderfarm_client_components. Jenkins builds full .exe and .deb installers out of it.
The main scripts are:
main/fileserver4_downloader.py (the process that downloads from the queue)
main/fileserver4_uploader.py (the process that uploads to the entrypoint)
main/fileserver4_nothreads.py (the process that serves metadata requests)

These are mostly flask or tornado HTTP services, which get called by the entrypoint.

All the entrypoint and cache setup configuration is written in puppet. It is in the repository deploy_manager, subdirectory bootstrap_linux_server/Head_setup.
Puppet modules of interest:
* metadata_proxy sets up the Metadata Cache Server
* openvpn
* raw_nginx_cache installs the File Cache Server
* seekscale contains the main install files (see especially master.pp for the entrypoint, and proxynet_proxy.pp for the cache server)
* smbproxy installs the smbproxy
* transferback_queue configures the queue to transfer files from the entrypoint to the gateway