EL2W: Extended Layer 2 Services for Bare-Metal Provisioning Over WAN

Thomas J. Hacker
Dept. of Computer and Information Technology
Purdue University
West Lafayette, Indiana USA
tjhacker@purdue.edu

Deepika Kaushal
GoDaddy.com LLC
Software Engineer
Kirkland, Washington USA
dkaushal1@godaddy.com

Zhiwei Chu
Dept. of Computer and Information Technology
Purdue University
West Lafayette, Indiana USA
chi32@purdue.edu

***Abstract*—Researchers are embracing deep learning in various interdisciplinary research domains, recognizing undeniable benefits offered by deep neural networks. However, in order to meet the substantial computational demands for processing deep learning models, researchers extensively rely on cloud servers. Nevertheless, the shared nature of cloud servers encourages research labs and facilities to establish private clouds, ensuring exclusive access to computational resources and safeguarding data privacy. Creating a private cloud from bare metal presents challenges with existing provisioning solutions. These solutions not only come with a set of complex installation and configuration steps but are also limited to a constrained local Ethernet broadcast domain for network loading, which may pose unforeseen difficulties and risks for researchers who do not specialize in computing. To address these issues, this paper introduces EL2W, Extended Layer 2 services to Wide area networks (WAN), a novel approach we developed following Infrastructure-as-Code (IaC) principles. EL2W aims to help automate the system installation procedure by reducing the repetitive configurations and setups using Infrastructure-as-Code based scripts and codes. In addition, EL2W can securely expand an Ethernet network's logical and functional extent beyond the current physical limitations of Ethernet layer 2 networks. We describe the implementation and architecture of a remote bare metal provisioning system built upon secure extended layer 2 networks. Experimental results demonstrate the capability of EL2W for establishing a secure layer 2 connection to provide essential bare metal provisioning services, as well as the effectiveness of a local proxy cache server to reduce the operating system loading time.**

***Index Terms*—bare metal, remote provisioning, extended layer 2 services, PXE boot, Infrastructure-as-Code**

I. Introduction

Deep learning research has made significant inroads into numerous interdisciplinary domains. Specialized hardware such as GPUs are essential to provide enormous computing power for processing deep neural networks. Although cloud servers such as community clusters provide the required computational resources, researchers may require exclusive or privileged access to computing resources, which cannot be easily satisfied with cloud servers. Additionally, labs with data security and privacy concerns may prefer to host their own data storage services and physically isolate the data from the public network. As a result, labs and facilities are looking towards a private cloud dedicated to themselves. However, bootstrapping the private cloud from bare metal can be troublesome for researchers from disciplines other than computing, such as civil engineering and mechanical engineering. To bootstrap physical servers or virtual machines, existing solutions for bare metal loading such as OpenStack [1] and MAAS [2] have certain limitations, including sophisticated installation processes and a local network connection requirement. The inability to extend the layer 2 services required for loading physical bare metals also limits the functionality of the services.

There are commercial and academic research efforts that seek to provide an efficient provisioning solution for bare metal systems. Popular bare metal provisioning projects provided by the open-source community and the commercial world include but are not limited to, Cobbler [3], OpenStack Ironic [1], Foreman [4], MAAS [2], and Razor [5]. A comprehensive comparison was provided by Chandrasekar and Gibson [6] in a study that compared six bare metal provisioning frameworks. These frameworks generally entail a sophisticated setup process and require the servers to be on the same physical network and bare metal installation images to be present on the local network. Other research also proposed several systems and solutions. Mohan and others proposed M2 [7], Malleable Metal as a Service, a diskless booting approach for provisioning bare metal systems.

These proposed frameworks and systems all aimed to provide an automated and rapid provisioning approach. However, to the best of our knowledge, we have identified the following needs and gaps from the existing solutions:

Simplified Services and Installation. The existing methods we have investigated all include complex installation and deployment processes. To illustrate, the OpenStack Ironic Service [1] has three major components and can be configured to communicate with up to six other OpenStack services [8]. High-level administrative skills are often needed when deploying such systems. A simplified installation mechanism is needed to quickly set up a provisioning service to boot and load local infrastructures for people to bootstrap bare metal systems with less effort.

Remote and Distributed. Second, related work by Mohan [7] and Daly [9] have proposed systems that support a diskless booting environment. The required layer 2 services are provided from local area networks (LAN). As a result, Mohan's and Daly's methods do not support bare metal provisioning beyond the local Ethernet broadcast domain. The physical lim-

itation of the layer 2 network results in a gap in provisioning layer 2 services over distributed broadcast domains. The gap can be further identified as the absence of a secure means to control the process of provisioning, publishing, and accessing layer 2 services in extended layer 2 networks. Additionally, the gap can also be recognized as the lack of mechanisms that would support the creation of new layer 2 services in a distributed Ethernet broadcast domain.

Rapid and Secure. In addition, other studies that include Mao [10], and Hao [11] suggested that the VM startup time is crucial for cloud elasticity. Omote [12] further stated that the same concept could also be extended to bare metal systems or instances. Therefore, methods to reduce the OS's initial deployment time become crucial in a distributed network architecture. Regarding security, if the remote connection between the bare metal servers and the provisioning service provider is not encrypted, unexpected miscellaneous services and resources may have the opportunity to communicate with bare metal systems during loading and potentially introduce cybersecurity risks. Consequently, fast and secure OS bootstrapping is critical for provisioning bare metal systems.

To address these needs and gaps, we propose *EL2W*, a straightforward, shareable, and secure approach for remotely provisioning bare metal systems or VMs from a trusted central source. The EL2W implementation fulfills the gaps as follows:

1) To achieve a simplified service and its installation process, we implemented the EL2W with only the essential services required for bare metal provisioning. In addition, the design of EL2W follows the Infrastructure as Code (IaC) [13] principle. We implemented EL2W using IaC tools to provide an automatic and simplified service deployment process. The adoption of IaC using Vagrant [14] results in a straightforward manner of deploying (*vagrant up*) and removing (*vagrant destroy*) our provisioning service with very few configurations.

2) To enable remote layer 2 connections across the extended Ethernet broadcast domain, we introduced the use of a local HTTP proxy server with a service that we named *EL2W ports*. We used EL2W ports for remote access to DHCP, BOOTP, and other required layer 2 and layer 3 services for establishing local infrastructures.

3) To ensure secure layer 2 connections over WAN, we implemented VXLAN tunnels over IPsec connections using a combination of Open vSwitch (OVS) [15] and strongSwan [16]. Furthermore, we enabled a two-factor Duo authentication [17] to protect the exchange of protected information during Infrastructure-as-Code based setup in the EL2W system.

4) To accelerate the OS booting process, we deployed an HTTP proxy server to provide a local cache for OS installation packages. By doing so, instead of downloading from a remote repository, the packages could be retrieved from a local cache server once the cache is populated. Test results indicate that caching can largely reduce OS boot and load time.

EL2W simplifies the creation of a secure cloud infrastructure and a secure wide-area Ethernet broadcast domain over which secure layer 2 services could be provided to remote clients. In this work, we used the implemented EL2W to provide a boot service for remote provisioning bare machines and VMs over an extended Ethernet broadcast domain.

The implementation of this work is used for remotely establishing a secured lab-scale private cloud for the VISER (VIsual Structural Expertise Replicator) project. VISER is built on the prior work from the ARIO project (Automated Reconnaissance Image Organizer) [18], [19]. VISER and ARIO aim to offer a post-disaster visual data classification service for large collections of image data management for civil engineers. The work described in this paper also relates to the M.S. thesis of co-author Kaushal [20], who implemented the remote booting concept separately.

The rest of this paper is structured as follows: section II summarizes and analyzes related work on bare metal provisioning and securely expanding an Ethernet broadcast domain. Section III depicts the architectural design and implementation details of EL2W. Section IV describes the use of EL2W for remotely provisioning bare metal systems or VMs. Sections IV and V discuss experimental results and lessons learned. Finally, conclusions are discussed in section VI.

II. Related Work

For the work described in this paper, we first investigated existing bare metal provisioning solutions and sought methods to extend layer 2 services beyond the local Ethernet domain.

A. Bare Metal Provisioning

We first looked into provisioning solutions offered from the commercial and open-source world.

OpenStack Ironic [1] manages systems using IPMI and provides a loading service that integrates with other OpenStack modules (e.g., Neutron for networking and Glance for VM image management). Likewise, Cobbler [3] offers provisioning along with other functions such as DNS and DHCP management, and configuration management orchestration. Furthermore, we investigated Foreman [4], which uses *smart proxies* with plugins as an abstraction layer to access remote services such as DNS, Puppet, DHCP, and TFTP. However, OpenStack, Cobbler, and Foreman all have sophisticated setup processes. For example, the main installation command for Foreman, *foreman-installer*, features over one thousand (1,151) command line options, 74 of which have additional "no" selective options [1]. We also examined MAAS (metal-as-a-service) [2]. MAAS operates on a data center scale using a tiered architecture and parallels the provisioning of deployment services across many MAAS service providers on a rack level. Although MAAS has a simpler installation and operations model, it does not facilitate the extension of an Ethernet broadcast domain to a distant network to provide layer 2 services. An evaluation was made by Chandrasekar and Gibson [6] to compare Emulab, Ironic, Crowbar, Razor, Cobbler, and MaaS across 14 different criteria, including the difficulty of installation and difficulty of maintenance.

We also found a web-accessible service *boot.netboot.xyz* [21] that can be chain loaded from iPXE during the boot

[1] Identified by running *foreman-install –full-help*

process, which provides a comprehensive menu of loadable OS and utility images.

Other recent related research work also proposed different bare metal provisioning systems. Mohan and others [7] presented a system named M2, Malleable Metal as a Service. M2 aimed to provision bare metal systems by replacing the target OS image downloading step with a local Ceph storage and exposed iSCSI targets. M2 also used iPXE to implement OS chain loading Omete and others [12] described another system called BMcast, an OS deployment system that implemented background copy and copy-on-read of boot images to reduce the instances' startup time.

We found that these bare metal provisioning systems have complex installation and operation procedures and require a high degree of skill and training to install, configure, maintain, customize, operate, and debug. These shortcomings can lead to poor *cybersecurity*, *auditability*, and *reproducibility* of the system installation, significantly increasing the complexity of installing, configuring, maintaining, and using these systems.

III. Architecture and Implementation

A. EL2W Ports

Our system uses Linux virtual network adapters to create local *EL2W ports* to provide a gateway to remote layer 2 services. We can assign names to the ports to reflect the service available through an EL2W port. For example, an EL2W port named *boot* could be the communication channel's attachment point for a boot service provided from a remote Ethernet network. EL2W ports can be used by a process or can be attached to a Linux network bridge or virtual machines to expand the broadcast domain and provide access to other internal or external system network adapters.

Our EL2W port approach is novel in the following ways:

1) EL2W ports introduce the concept of *publishing* and *unpublishing* layer 2 services to control access and allow the sequential staging of access to running services.

2) We use Infrastructure-as-Code (IaC) techniques to allow users to easily create, customize, audit, and reproduce their EL2W infrastructure.

3) We allow for extended Ethernet broadcast domains over a WAN to provide a platform that could be used for developing and testing new layer 3 protocols. The platform can be independent of legacy IP awareness within the new protocol.

4) We use a hub and spoke approach for the EL2W system that can automatically use spoke level caching on a single spoke via Squid proxy cache software [22] of information downloaded through a central EL2W hub over HTTP that could be used to speed up recurring actions (e.g., loading many nodes in a cluster).

EL2W ports can be used to create, export, and publish a layer 2 service on an EL2W hub through an EL2W spoke on the Ethernet broadcast domain attached to the EL2W spoke.

B. Architecture of EL2W Hub and Spoke

A conceptual overview of our EL2W system is shown in Figure 1. An EL2W hub *Sun* is connected via secured IPsec-based VXLAN tunnels to EL2W spokes *Venus*, *Earth*, and *Mars* using Open vSwitch (OVS) virtual switches on the hub and spoke nodes. The role of a central EL2W hub is to provide services while the remote EL2W spokes are to provide remote intermediate "jump hosts" to act as a local gateway to access services on the central EL2W hub. External bare metal systems or virtual machine instances can join an extended Ethernet broadcast domain through the spoke to access layer 2 or layer 3 services on the central EL2W hub by attaching to the external (or internal) network interfaces that are connected to the OVS virtual switch on the service spoke.

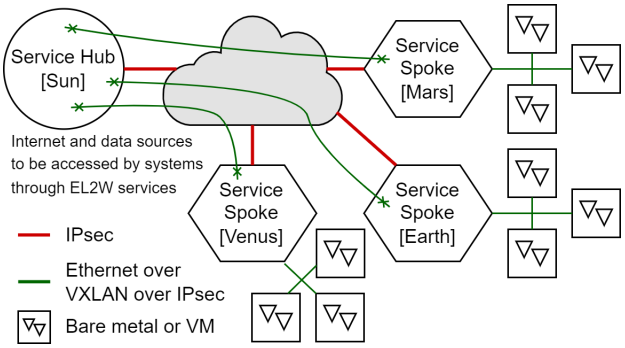

Fig. 1. EL2W Service Overview

Figure 2 shows the architecture of an EL2W hub, *Sun*. The hub Sun can be quickly constructed on demand using Vagrant [14] with a Vagrantfile and configuration scripts. We used an Ubuntu server with VirtualBox and Vagrant to establish a baseline VM for the EL2W hub and completed the node installation using the scripts we developed to load and configure the EL2W hub. The IaC approach allows a complete reproducibility of the EL2W hub and allows others to use the same Vagrantfile and infrastructure creation scripts to create and customize their own EL2W hub.

Starting from the bottom of Figure 2, other than the external interface managed for Vagrant access, Sun's external interfaces are assigned to the firewall zone *Public*: one interface is bridged to the server's physical interface with a publicly accessible external IP address, and a second interface is assigned the hub Sun's IPsec IP address **H** with a unique subnet (that differs from the public IP address subnet) that is attached to the VirtualBox internal network. This allows Layer 3 (IP) UDP/TCP port access for IPsec, VXLAN, ssh, and rsync configured with Duo 2-factor authentication (2FA) (in our implementation). The internal virtual Linux interface *boot*, which is used as an EL2W port interface, is associated with a separate firewall zone for interface *boot* with open ports for the Layer 3 services used by the boot service, which are: DHCP, DNS, HTTP, NFS, Squid, and TFTP.

Above the firewall in Figure 2, take the leftmost link as an example, an EL2W spoke's IPsec address (**X**) and an EL2W hub IPsec address (**H**) formulate an IPsec tunnel pair (**X**|**H**) to create separate IPsec connections. StrongSwan [16] is used to create IPsec connections among all EL2W hosts. A VXLAN tunnel with an assigned ID (VNID **A**) can then be established

over the IPsec link between the EL2W hub and spokes. This can be combined with the IPsec tunnel pair (**A, X**|**H**) to differentiate EL2W spoke's links on the hub.

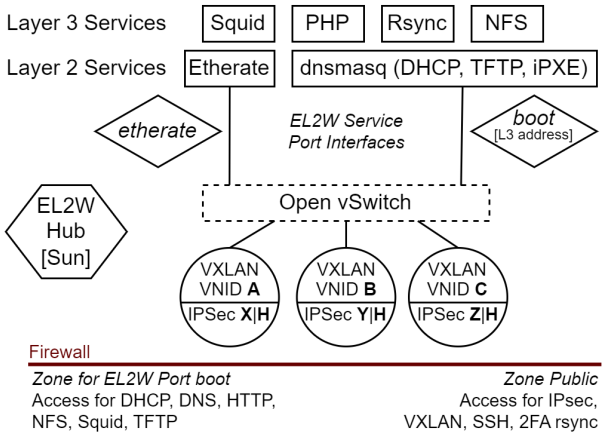

Fig. 2. EL2W Service Hub (Sun)

To provide connectivity and a common Ethernet broadcast domain between the EL2W hub and EL2W spokes, we use an OpenVSwitch (OVS) virtual switch to link the interfaces. To provide a communication path from the OVS switch to services, the additional named virtual interfaces (*etherate* and *boot* in Figure 2) can be attached to layer 2 services such as *dnsmasq* or *Etherate* [23]. Moreover, if a layer 3 IP address is associated with the named virtual interface, then layer 3 services can be accessed on the service hub over the secure IPsec channel from a separate EL2W IP address domain (such as 192.7.7.0/24). This IP address space can be controlled using routing and firewall rules. Moreover, the EL2W hub can be placed behind a firewall to provide an additional layer of network defense. In this work, we used a pfSense firewall [24] with a virtual external IP address to manage incoming traffic to the EL2W hub server.

On the EL2W Hub, we created two different services: a **Boot** service for remote provisioning bare metal systems or VMs via iPXE, and an **Etherate** service based on the Etherate software tool developed by Bensley et al. [23] that provides an endpoint for testing the Ethernet link from EL2W spokes[2]. The **Boot** service is built on a combination of layer 2 and layer 3 services including dnsmasq (offering DHCP, TFTP, iPXE), NFS, PHP, and Squid. Note that since the *etherate* interface provides access only to a layer 2 service, no IP address nor IP firewall zone is assigned. In order to use a local HTTP proxy for downloading necessary packages when loading operating systems, we modified the iPXE code [25] with a patch from Chirossel [26]. During the hub setup procedure, the hub generates the IPsec Certificate Authority (CA) certificates and private keys for the spokes. Subsequently, during the spoke setup process, the certificate and corresponding private keys are transmitted from the hub to the spoke via rsync, which is configured with Duo 2FA. The employment of Duo 2FA introduces an additional layer of security by mandating user confirmation through the Duo application on a smartphone during the key transfer process. After an EL2W hub is established, EL2W spokes can be quickly created using Vagrant in the same manner as the creation of the EL2W hub. Figure 3 shows the architecture of an EL2W spoke, Mars.

[2]We modified the Etherate software tool to work around problems we experienced when using Etherate to test the layer 2 links.

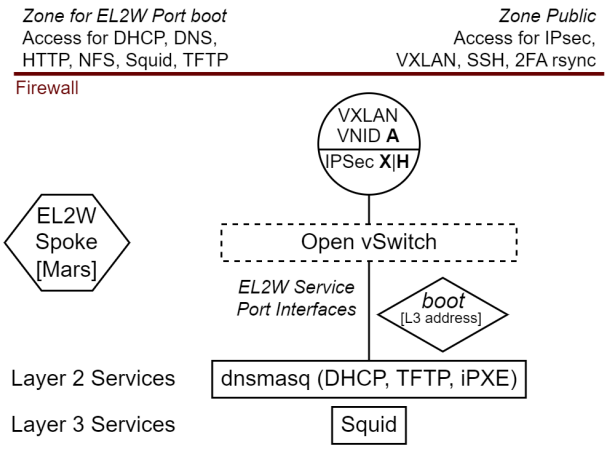

Fig. 3. EL2W Service Spoke (Mars)

From the top of Figure 3, a firewall on the spoke provides layer 3 protection for the *Public* zone and the *boot* internal network interface. Similar to the EL2W hub, a spoke has a VirtualBox adapter for local access managed by Vagrant; a VirtualBox internal network for connecting bare VMs or a bridged adapter for connecting bare metal systems; and another VirtualBox internal network with the EL2W spoke's IPsec endpoint IP address. Below the firewall, an IPsec connection is established with the IPsec link **X**|**H**. The spoke needs the public IP address of the EL2W hub and a unique IPsec IP address to establish an IPsec connection. Then, a VXLAN connection with pre-configured VNID **A** can be set up with the EL2W hub to create a layer 2 tunnel (**A, X**|**H**) to pass Ethernet frames. An Etherate client on the newly created VXLAN port can be used to access the **Etherate** service on the EL2W hub to test the connectivity of the layer 2 tunnel. To provide local DNS services (which we use to manage the resolution of local IP addresses for spoke or hub hostnames), a local dnsmasq service runs on the spoke.

The role of an EL2W spoke includes a proxy server and a jump host. As a proxy server, the spoke serves as an intermediary to access the services provided by the EL2W hub. As a jump host, the spoke provides limited secure access to the extended Ethernet broadcast domain accessible via the EL2W port and the layer 3 services protected by firewalls on the EL2W spokes and hubs.

IV. Using EL2W to Remotely Load an OS

This section describes a use case we implemented using the EL2W system for remote provisioning bare metal systems (including bare VMs). We illustrate this capability through the

example of a remote booting and loading service exported from an EL2W hub through an EL2W spoke to the bare-metal systems (physical servers or VMs) connected to the same Ethernet broadcast domain of the EL2W spoke.

This example is motivated by the need to simplify the provisioning of bare metal systems or virtual machines within a laboratory or at a location where the availability of skilled personnel is limited. This would be essential if a system is corrupted or compromised and needs to be rebuilt quickly from a known source for packages. This example is also motivated by a use case in which many local systems need to be loaded. As described previously in this paper, the gaps that make this difficult today without the use of our EL2W system are: 1) the level of necessary skill or knowledge of system administration; 2) the need for a controlled central hub for providing installation images and configuration that could be leveraged for higher-level services; and 3) the need for reproducibility and simplifying the reconstruction of compromised infrastructure.

A. Setting Up an EL2W System

To create and publish a remote boot and loading service, we first create an EL2W hub using the hub's Vagrantfile and configuration scripts. With an EL2W hub established, we can then use a spoke Vagrantfile and configuration scripts to create one or more EL2W spokes that use secure IPsec links to the EL2W hub and published EL2W ports accessible through the spokes. The example we describe in this section needed and used only one spoke.

There are three configuration scripts that built the services on the base virtual box images: *el2w_create* is used to create the named EL2W endpoint and *boot_service* installs, configures, and publishes the remote booting and loading service. The third script *etherate_service* is used to start the Etherate testing service on the EL2W hub. The *etherate_service* script only executes on the EL2W hub while the other two scripts will run on both hub and spokes.

The parameters of the *el2w_create* script include the role (hub or spoke), the IPsec adapter name and address, the VXLAN adapter name, connection name, and VNID, and the spoke endpoint name.

The *boot_service* script then creates the **Boot** service. The parameters include the role (hub or spoke), EL2W service port name, OVS bridge name, a private IP address for boot service, IP address for a local squid proxy, and local, boot, and root DNS addresses. This script installs necessary software packages, creates configuration files, configures the OVS bridge, modifies the interface MTU (Maximum Transmission Units) size, and starts the DHCP, iPXE, TFTP, HTTP, and Squid processes. The script also retrieves and prepares OS images for remote provisioning, configures the firewall rules, copies the kickstart files for automated installation, configures and starts the NFS service, and publishes the layer 2 **Boot** service.

The *etherate_service* script sets up the Etherate test service for the EL2W hub. The script accepts the parameters describing the role (hub or spoke) and the name of the OVS bridge to which the EL2W **Etherate** service will be attached.

TABLE I
EL2W HUB AND SPOKE BUILD TIMING RESULTS (IN MIN.)

Role	Build Time	Server Specs
Hub	22.67 ± 1.57	2 × Intel(R) Xeon(R) E5-2643 32GB RAM + 2TB HDD
Spoke	7.74 ± 0.16	Intel(R) Core(TM) i7-6700 16GB RAM + 500GB SSD

The time spent on setting up an EL2W Hub and Spoke using *vagrant up* (with VirtualBox) was recorded. We used two timing test scripts for running without human interaction, calculating the average time and the sample standard deviation (SD). Table I shows the server specs and the timing results. The results are collected and calculated from 10 trials. We used one Vagrantfile for the hub and one for the spoke with internal parameters. The Vagrantfile was configured to create a base Rocky 8 VM, update the OS, install basic security, set up network adapters, and perform the initial firewall configuration. A local directory, which contains the configuration scripts and files needed for the second stage of installing the hub and spokes, is mounted into the VM. The same Vagrantfile and configuration scripts were used across each trial as a control variable.

All the experiments were conducted in the same network environment, whereas the download speed is still considered an independent variable. Therefore, when building the EL2W hub, to reduce the timing variance caused by downloading large OS images from a remote repository, the ISO files for the OS distributions were also downloaded beforehand and mounted via the Vagrantfile. Furthermore, to eliminate the need for user interaction during the loading process, when setting up the EL2W spoke, the hub's authentications were disabled in the configurations to bypass the Duo manual confirmation step.

The EL2W hub's server is a Dell PowerEdge R620 1U rack server with 2 Intel Xeon E5-2643 CPUs, 32 GB RAM, and 2 TB HDD storage. Meanwhile, the EL2W spoke Mars is on a Dell OptiPlex 7040 desktop PC with an Intel Core i7-6700 CPU, 16 GB RAM, and 500 GB NVMe SSD. The hub and spoke servers use Ubuntu 20.04.5 as the host operating system. In addition, VirtualBox 6.1.42 and Vagrant 2.3.4 are installed on both hub and spoke's host machines. The structural network speed limitation to the hub was 1 Gbps (limited to a 1 Gbps NIC on the pfSense firewall), and the limit to the spoke was 200 Mbps (ISP limitation). Under these conditions, the average time for setting up the EL2W hub was 22.67 minutes, with a sample standard deviation of 1.57 minutes. The average completion time for building the Mars spoke was 7.74 minutes, with a 0.16 minutes sample standard deviation.

B. Example: Establishing a Boot Service

Once the EL2W hub Sun is up and running, we can then establish an EL2W service spoke (i.e., Mars) using *vagrant up* with a parameterized Vagrantfile and private Duo keys for rsync to communicate with the EL2W hub.

To access the boot service provided by an EL2W spoke, a virtual machine running on the same hardware as the EL2W spoke, or an external bare metal system can be attached to an external switch or connected to an Ethernet port on the spoke that is also attached to a network adapter on the EL2W spoke virtual machine (within the same Ethernet broadcast domain). This provides a network path to access the EL2W services from bare metal systems or VMs. The internal network adapter on the EL2W spoke is connected to the OVS virtual switch on the spoke to provide a layer 2 path to the EL2W hub.

The process of establishing an EL2W spoke uses the service scripts *el2w_create* and *boot_service*. As a part of building the layer 2 path, a connectivity test service using Etherate [23] is running on the EL2W spoke to verify the path is functioning before building the boot service on the EL2W spoke.

To use the remote boot service, a bare system is configured to perform a network boot (supported by both BIOS and UEFI systems). The initial DHCP request is satisfied by the dnsmasq server running on the remote EL2W hub (Sun) (which also provides the IP address with the DHCP response for the Mars spoke dnsmasq DNS service), boot parameters, and a PXE boot script that is sent to the bare system. The PXE script displays several loading options to the user. We currently provide CentOS 7, CentOS 8, Rocky, TrueNAS, FreeNAS, and pfSense, as shown in Figure 4.

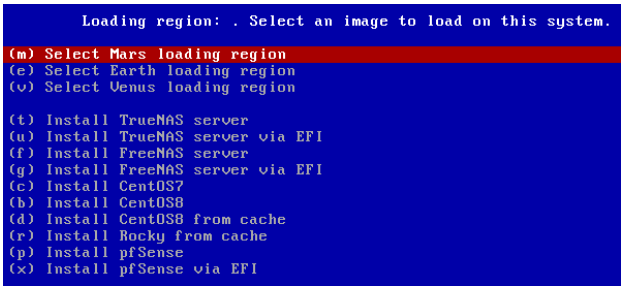

Fig. 4. EL2W Boot Service Boot Options iPXE Menu

The PXE script sent to the bare system is chain loaded using two stages. The first stage uses a PXE script embedded into the first *undionly.pxe* (*ipxe.efi* for UEFI systems) executable built automatically via IaC on the EL2W hub. This patched iPXE we used supports HTTP proxies which allow the remote systems to exploit the Squid cache on a single EL2W spoke to improve performance for subsequent OS loads from HTTP. The first PXE script sets the HTTP proxy address in the PXE environment and then requests a PHP script on the EL2W hub that creates a secondary PXE script. The PXE code generated and returned to the bare system from the EL2W hub builds the menu presented to the user. The menu shown in Figure 4 allows the user to select a loading region and an OS to be provisioned on the system. The PXE script leverages the iPXE option variables *user-class* to set the loading region (Mars, Earth, or Venus), *http-proxy* to set the Squid cache HTTP proxy address, and *vendor-class* to convey the system architecture (BIOS or UEFI).

This approach returns the user-selected parameters to the EL2W hub via a PXE *imgfetch* command and invokes another PHP script (if needed) to set up the DHCP and iPXE options required to boot bare systems from a specific NFS volume for FreeBSD-based OSes from the EL2W hub. Rocky and CentOS booting is simpler and only requires an image load from a remote repository. We also created a local copy of the distribution media for CentOS and Rocky on the hub that could be distributed via HTTP from the EL2W hub. Since the images and packages for CentOS and Rocky are remotely retrieved using HTTP, they can be stored as the Squid cache on a single EL2W spoke to speed up subsequent loadings.

In this work, the EL2W spoke machine was located approximately a 1.3-mile driving distance from the hub server, and the spoke machine was connected to a different network than the hub. After the EL2W infrastructure is built and set up, we can test the remote provisioning functionality the EL2W **Boot** service provides. VirtualBox VMs are created on a Windows laptop as bare VMs for measuring BIOS PXE boot. The Windows laptop has an Intel Core i7-12700H, 64 GB RAM, and a 2 TB PCIe 4.0 SSD. The bare metal for measuring UEFI is a Dell OptiPlex 7010 desktop PC with an Intel Core i7-3770, 16 GB RAM, and 256 GB SATA SSD. The bare VMs are connected to the spoke via a bridge network adapter in VirtualBox, and the bare metal desktop is connected via the onboard Ethernet port. Both bare systems are connected to the spoke via a 1 Gbps connection, and the physical Ethernet port is the same bridge adapter defined in the Vagrantfile while setting up the EL2W spoke. During our development, we tested both UEFI and BIOS in VirtualBox. The measurements shown in Table II for VM loading are for BIOS only.

The time in minutes needed to load CentOS 7 and Rocky in different scenarios is shown in Table II. For each OS and platform scenario, the timing results of the different methods were collected from three independent trials, and the average and sample standard deviation (Avg.$\pm$SD) of the timing results were calculated. Different methods were used when loading CentOS 7 and Rocky for comparing direct HTTP downloading from the hub and Squid cache on the spoke. As for HTTP methods, we measured the provisioning time using direct downloads from a remote public repository (HTTP (Remote) for loading CentOS 7) and the EL2W hub's HTTP server (HTTP (Hub) for loading Rocky). Using an HTTP server on the EL2W hub to self-host the OS images can not only isolate the loading process but also enables the capability for loading and booting customized OS images for specialized internal use. After the first load, the files will be cached using Squid on the spoke for subsequent bare system provisioning. The CentOS 7 and Rocky provisioning process was configured for the measurements to eliminate user interaction during the loading process. To remotely load TrueNAS, which is based on FreeBSD, the EL2W hub exported the TrueNAS file root directory through NFS. By selecting from the iPXE menu, the TrueNAS root directory was mounted to the client via the extended NFS service from Spoke's *boot* port. Then, a *pxeboot* file from the TrueNAS boot directory was loaded first.

The *pxeboot* file then finished the rest of the TrueNAS loading using the NFS-mounted directory. The loading process of TrueNAS was not fully automated and required user interactions.

TABLE II
EL2W REMOTE OS LOADING TIMING RESULT (IN MIN.)

OS	Method	BM (UEFI)	VM (BIOS)
CentOS 7	HTTP (Remote)	5.84 ± 0.28	4.71 ± 0.12
	Spoke Squid	4.79 ± 0.03	3.7 ± 0.19
Rocky	HTTP (Hub)	5.46 ± 0.21	4.43 ± 0.08
	Spoke Squid	4.78 ± 0.03	4.09 ± 0.07
TrueNAS	Hub NFS Share	27.57 ± 0.05	28.46 ± 1.04

The loading time for CentOS 7 and Rocky started from selecting the OS from the iPXE menu (shown in Figure 4) to the time when the OS login page appears. Using HTTP (Remote) method (first access to the OS DVD image from a site remote from the hub), the time needed for provisioning CentOS 7 on an actual physical bare metal (BM) and bare virtual machine (VM) was 5.84 ± 0.28 minutes and 4.71 ± 0.12 minutes, respectively. Using the cached data from the Squid cache, the time was reduced to 4.79 ± 0.03 minutes for bare metal and 3.7 ± 0.19 minutes for bare VM. As for Rocky, using the hub's HTTP server (with the OS DVD image already on the hub) required 5.46 ± 0.21 minutes to provision a bare metal system and 4.43 ± 0.08 minutes to provision bare VMs. Using the spoke's Squid cached data to load Rocky on a bare metal system took 4.78 ± 0.03 minutes, while on bare VMs, the time was 4.09 ± 0.07 minutes. The results show an overall time reduction of approximately 15% from using the spoke Squid cache after the first load that didn't use the Squid cache. The TrueNAS provisioning time started at the same phase but ended at the "TrueNAS installation succeeded" prompt. As a result, the remote loading of TrueNAS on bare metal and bare virtual machines used 27.57 ± 0.05 minutes and 28.46 ± 1.04 minutes, respectively.

Our experiments and measurements showed that it was possible to remotely load a CentOS 7, Rocky, or TrueNAS (based on FreeBSD) based OS from a remote server using our EL2W approach. As shown in Table II, it required a reasonably short time to load CentOS 7 and Rocky remotely. Moreover, we found that using a local Squid cache on the spoke could significantly reduce the time needed to load CentOS 7 and Rocky. The difference in time between loading BM versus a VM for CentOS 7 and Rocky could possibly be due to the differences in computer hardware used for the bare metal system and the laptop used for the VM. The BM system has an older processor than the laptop used for the VM and a SATA connection to the drive compared to a PCIe connection for the laptop. We did not investigate the underlying performance differences between loading a BM versus a VM, all the timing results were collected from loading one single bare metal with one operating system at a time to control variables. The hub and spoke architecture design theoretically supports multiple spokes existing at different physical locations at the same time. However, this will require sufficient bandwidth to support the network traffic of multi-connections. In this work, the viability and scalability of our EL2W was not investigated.

In co-author Kaushal's implementation of the concept (described in her thesis [20]), the efficiency of a Squid cache was demonstrated by mimicking delays corresponding to different regions across the globe. Kaushal compared the system average loading time of 11 consecutive runs using the framework with and without a Squid cache. The result indicated that without Squid, the loading time increased linearly with the imposed network delays increasing from 0 ms to 67 ms (using *NetEm*). Whereas, with a Squid cache, the time required to load the system stayed similar despite the different delays. As the network diameter increased, Kaushal witnessed a much more significant reduction in loading time as subsequent loads can exploit the spoke squid cache. Thus, if multiple bare systems are loaded with CentOS 7, the successive loads will require much less time with less traffic to the EL2W hub than without a Squid cache after the first load.

After the bare systems are loaded, they are provided with an private IP address (192.7.7.0/24). The node can then securely access hosts across the EL2W network within the private IP space. IP addresses for external destinations are routed to the EL2W hub through the private IP space and protected by an outward-facing firewall on the EL2W hub. Any other external firewalls (such as pfSense) can be set up in front of the EL2W hub to provide another layer of protection.

V. DISCUSSION AND LESSONS LEARNED

Building this infrastructure based on IaC principles took considerable effort on several fronts. In this section, we interpret our knowledge gained and our findings while conducting the experiments.

A. System Reproduction and Maintenance

First, reproducibility is essential. The Vagrantfiles and scripts must work every time, and configuration issues (such as IP masquerading and SELinux policies) often require investigation and resolution. The concern about this IaC approach is that with many "moving parts" required to make this work, as the software packages on which the EL2W system relies are upgraded, some functionality will inevitably stop working and need to be updated. The IaC approach's benefit is that by using software revision control, we can track these changes over time and see what has changed as the infrastructure evolves.

B. VXLAN over IPsec Performance

During the experiments, we discovered that the combined use of IPsec and VXLAN was causing packet fragmentation issues. To address this, during the establishment of the boot service, we had to manually change the MTU size of the *boot* interface on the EL2W hub.

During our experiments, we found that the interfaces' default MTU sizes were 1,450 bytes, and using the *tracepath* command measured the PMTU size between the spoke and hub IPsec tunnel to be 1,438 bytes. During our tests, we eventually changed the MTU size of the VXLAN interface (*boot*)

to 1,388 bytes on both EL2W's spoke and hub to achieve the spoke's connected network bandwidth (212 Mbits/sec, measured using iperf3 [27]).

C. NFS over WAN Performance

Furthermore, we also explored the long FreeBSD-based system provisioning time. We found that NFS performance was affected by the round-trip time (RTT) between the NFS client and server [28]. The FreeBSD-based system provisioning process uses NFSv3 to transfer the kernel and other package files. We observed that during the FreeBSD-based OS loading process, the client would send an acknowledgment back to the server for each packet sent from the NFS server to the bare system. In our experiments, the average RTT calculated from 1000 pings between the EL2W hub and spoke was 24.338 ms.

We tested one method to boost NFS performance over a high RTT connection: increasing the *read_ahead_kb* value of the NFS client [29] in Linux. To experiment, we mounted a shared directory on the hub to the spoke via NFSv3 using the UDP protocol. We increased the pre-fetch file size from the default of 128kb to 153600kb, and the NFS transfer speed from hub to spoke increased from 8.1 MB/s to 17.8 MB/s.

VI. Conclusion

This paper described the design and implementation of EL2W, a system we built using Infrastructure-as-Code tools and principles. EL2W provides a secure extended layer-2 Ethernet broadcast domain that can be used for a remote booting service on a small scale such as university research labs. We demonstrated the usability of our approach with examples of remote booting and loading operating systems on both bare virtual machines and actual physical bare metal machines. Our work provides the basis for aiding efforts to provision systems that form the foundation of a private small-scaled distributed computing infrastructure.

Acknowledgment

The work described in this paper is supported by National Science Foundation (NSF) award OAC-1835473. The opinions, findings, and conclusions or recommendations expressed are those of the author(s) and do not necessarily reflect the views of the NSF.

References

[1] "Bare Metal Service Installation Guide," Available at https://docs.openstack.org/ironic/latest/install/index.html, accessed March 2023.

[2] "MAAS Documentation," Available at https://maas.io/docs, accessed February 2023.

[3] "Welcome to Cobbler!" Available at https://cobbler.github.io/, accessed March 2023.

[4] "What is Foreman?" Available at https://www.theforeman.org/introduction.html, accessed March 2023.

[5] "Provisioning with Razor," Available at https://www.puppet.com/docs/pe/2019.7/provisioning_with_razor.html, accessed February 2023.

[6] A. Chandrasekar and G. Gibson, "A Comparative Study of Baremetal Provisioning Frameworks," *Parallel Data Laboratory, Carnegie Mellon University, Tech. Rep. CMU-PDL-14-109*, December 2014.

[7] A. Mohan, A. Turk, R. S. Gudimetla, S. Tikale, J. Hennesey, U. Kaynar, G. Cooperman, P. Desnoyers, and O. Krieger, "M2: Malleable Metal as a Service," in *2018 IEEE International Conference on Cloud Engineering (IC2E)*, 2018, pp. 61–71.

[8] "Bare Metal service overview," Available at https://docs.openstack.org/ironic/latest/install/get_started.html, accessed May 2023.

[9] D. Daly, J. H. Choi, J. E. Moreira, and A. Waterland, "Base Operating System Provisioning and Bringup for a Commercial Supercomputer," in *2007 IEEE International Parallel and Distributed Processing Symposium*, 2007, pp. 1–7.

[10] M. Mao and M. Humphrey, "A Performance Study on the VM Startup Time in the Cloud," in *2012 IEEE Fifth International Conference on Cloud Computing*, 2012, pp. 423–430.

[11] J. Hao, T. Jiang, W. Wang, and I. K. Kim, "An Empirical Analysis of VM Startup Times in Public IaaS Clouds," in *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*, 2021, pp. 398–403.

[12] Y. Omote, T. Shinagawa, and K. Kato, "Improving Agility and Elasticity in Bare-Metal Clouds," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 145–159. [Online]. Available: https://doi.org/10.1145/2694344.2694349

[13] K. Morris, *Infrastructure as code: managing servers in the cloud*. O'Reilly Media, Inc., 2016.

[14] "Vagrant Documentation," Available at https://developer.hashicorp.com/vagrant/docs, accessed March 2023.

[15] "Open vSwitch," Available at https://github.com/openvswitch/ovs, accessed June 2023.

[16] "StrongSwan," Available at https://strongswan.org/, accessed March 2023.

[17] "Getting Started with Duo Security," Available at https://duo.com/docs/getting-started, accessed March 2023.

[18] C. M. Yeum, S. J. Dyke, B. Benes, T. Hacker, J. Ramirez, A. Lund, and S. Pujol, "Postevent Reconnaissance Image Documentation Using Automated Classification," *Journal of Performance of Constructed Facilities*, vol. 33, no. 1, p. 04018103, 2019. [Online]. Available: https://ascelibrary.org/doi/abs/10.1061/\%28ASCE\%29CF.1943-5509.0001253

[19] S. J. Dyke, X. Liu, J. Choi, C. M. Yeum, J. A. Park, M. Midwinter, T. J. Hacker, Z. Chu, J. Ramirez, R. Poston, M. Gaillard, B. Benes, A. Lenjani, and X. Zhang, "Learning from Earthquakes Using the Automatic Reconnaissance Image Organizer (ARIO)," *Proceeding of the 17th World Conference on Earthquake Engineering*, September 2021. [Online]. Available: https://par.nsf.gov/biblio/10308795

[20] D. Kaushal, "Bootstrapping a Private Cloud," Master's thesis, Purdue University Graduate School, 6 2020. [Online]. Available: https://hammer.purdue.edu/articles/thesis/Bootstrapping_a_Private_Cloud/12576611

[21] A. Messerli, "Netboot.xyz," Available at http://netboot.xyz, accessed March 2023.

[22] "Squid caching software," Available at http://www.squid-cache.org, accessed March 2023.

[23] J. Bensley, B. Kite, B. de Montis, and C. Lucas, "Etherate," Available at https://github.com/jwbensley/Etherate, accessed March 2023.

[24] C. M. Buechler and J. Pingle, "pfsense: The definitive guide," *Reed Media Services*, 2009.

[25] "ipxe - open source boot firmware," Available at https://ipxe.org/ and https://github.com/ipxe/ipxe.git, accessed March 2023.

[26] O. Chirossel, "[ipxe-devel] HTTP proxy support," Available at https://lists.ipxe.org/pipermail/ipxe-devel/2016-December/005292.html, accessed March 2023.

[27] "iperf3 Network Bandwith Measurement Tool," Available at https://github.com/esnet/iperf, accessed March 2023.

[28] D. Greenfield, "Network file system (nfs)," Available at https://www.silver-peak.com/applications/nfs, September 2014, accessed February 2023.

[29] "RHEL8: NFS streaming read performance is slower after kernel update," Available at https://access.redhat.com/solutions/5953561, January 2023, accessed March 2023.