SDP Memo 29: Integration Prototype Build and Deployment Infrastructure Design

Document Number.....SDP MEMO 29
Document Type.....MEMO
Revision.....01
Author.....Iain Emsley
Release Date.....2016-12-15
Document Classification..... Unrestricted
Status.....Released

Lead Author	Designation	Affiliation
Iain Emsley		Oxford e-Research Centre, University of Oxford
Signature & Date:	**Signature:** Iain Emsley (Dec 15, 2016) **Email:** iain.emsley@oerc.ox.ac.uk	

ORGANISATION DETAILS

Name	Science Data Processor Consortium
Address	Astrophysics Cavendish Laboratory JJ Thomson Avenue Cambridge CB3 0HE
Website	http://ska-sdp.org
Email	ska-sdp-pa@mrao.cam.ac.uk

SDP Memo Disclaimer

The SDP memos are designed to allow the quick recording of investigations and research done by members of the SDP. They are also designed to raise questions about parts of the SDP design or SDP process. The contents of a memo may be the opinion of the author, not the whole of the SDP.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

1. Introduction

This document outlines the build and deployment servers in Oxford used for the Integration Prototype for the Square Kilometre Array (SKA) Science Data Processor (SDP). These machines install onto a cluster of test machines, each of which is separate.

This document outlines the notes of the conversation had with Ben Mort about the requirements for developing the system and the redevelopment of an existing service. This service installed both Ubuntu and Centos images and used Puppet for deployment. As the software requirements became standardised and understood, this is replaced by the system described in this document.

The build and deployment machines are installed on two different servers. This system provides a "Metal as a Service" (MAAS) deployment, where physical machines are treated as a cloud. This allows the user to install the required software onto an environment that uses the defined operating system of the SKA. Unlike Cloud, MAAS provides the infrastructure for user controlled dynamic provisioning. Instead of providing a complete environment, this service provides the infrastructure to complete the environment. This allows the machines to be used for a variety of tests with minimal reliance upon an operations team and for the use of DevOps processes to manage the machines.

Our aim is to provide a clean, basic infrastructure for prototyping interfaces and workflows that may be used for the SDP. As part of this, we provide Docker to allow existing users to install their own work on to a set of services.

As a by-product, the assumptions of the Software Repository Database can be tested.

Our build system deploys an Ubuntu 16.04.1 LTS netboot base image with a standard user, which has sudo rights. It installs docker from the Docker website and adds the user to the docker group so that sudo is not required to run the service.

The Docker container system is used within the deployment system. This follows practice in LOFAR and Meerkat where Docker is used and echoes the assumptions made for the Software Repository Database (C 1.1.3.2.4). Another deployment option is currently Puppet, which ALMA uses, and CERN has used, though CERN appears to be moving towards Docker for its cloud installations.

The requirements are outlined before we describe the system architecture and the services. We consider some of the pierformance issues that we have discovered before commenting on future work and the concluding.

The code has been stored in the SKA SDP Github repository:
https://github.com/SKA-ScienceDataProcessor/ip_deploy_build

2. Requirements

- Install a given machine with an operating system.
- Ensure that the operating system has a basic set of services, such as Docker.
- The build should be automated, including a user with sudo rights and networking.

Developed after the Integration Prototype meeting:

- The user should be able to request a software stack to install, such as Docker or OpenStack.

3. System Architecture

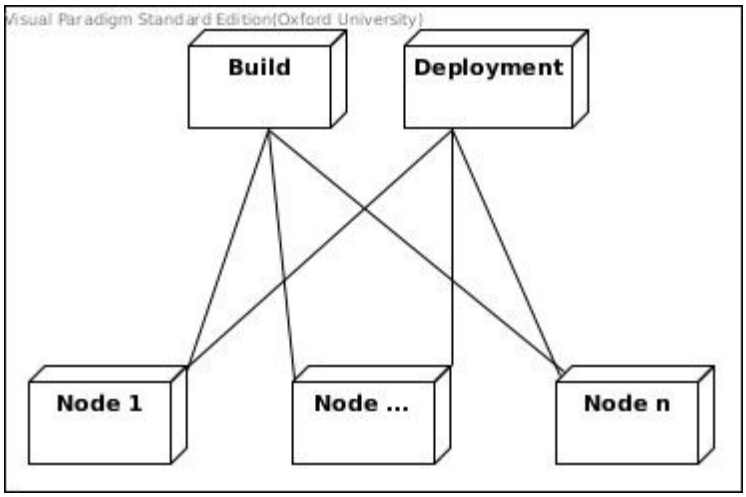

Figure 1: Deployment Diagram of the Build and Deployment Servers

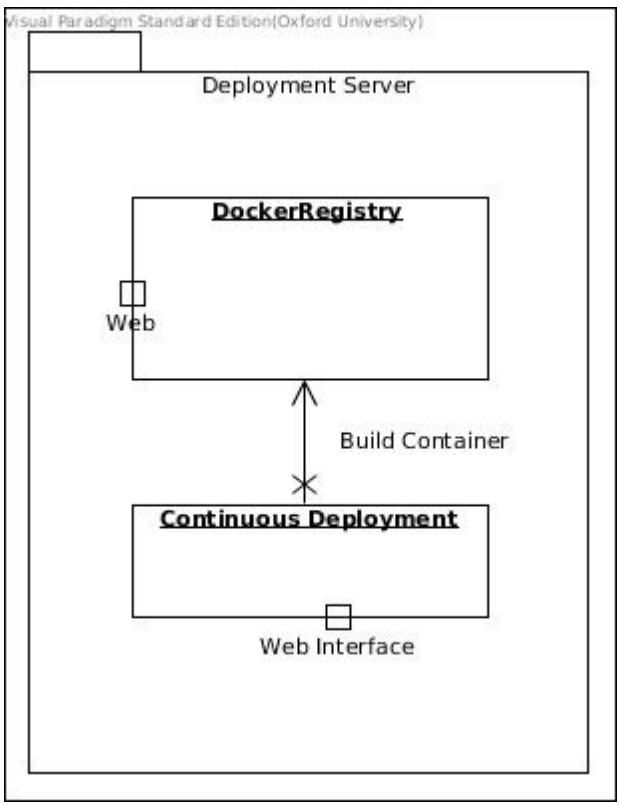

Figure 2: Component Diagram of the Deployment Server

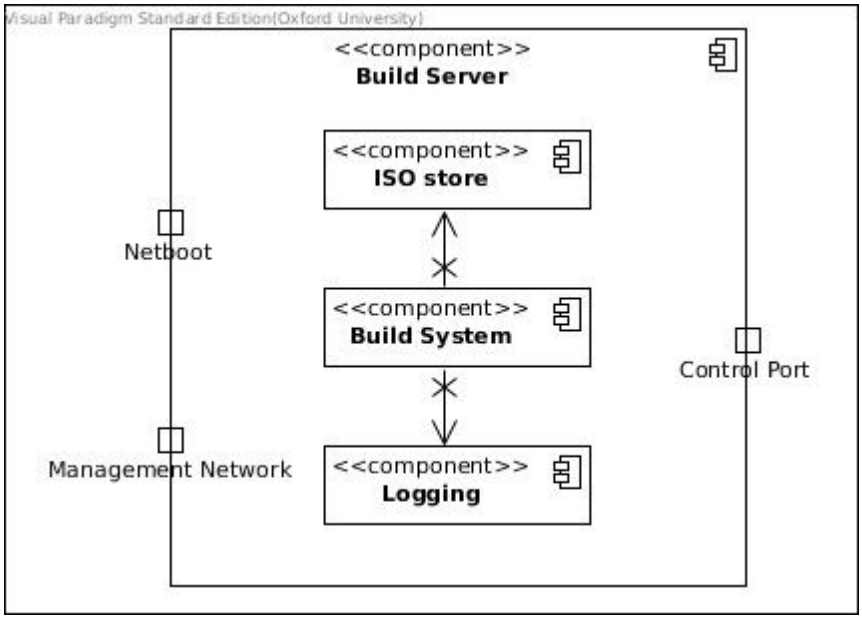

Figure 3: Component Diagram of the Build Server

4. Build Server

The build server, shown in Figure 2, contains the scripts and services to control and complete the installation of the target machines. This server is explicitly for the installation of the operating system and the initial software stack, such as Docker and OpenStack.

The build and the deployment servers are on separate machines. Although the Oxford system is small, it would be inefficient to build them together where the netboot ISO image is delivered with containers on the same network connection, so creating congestion.

In this section, the method of controlling the servers is discussed and the configuration for the files.

4.1. Controlling Servers via Lights Out Management

This is currently controlled by the Intelligent Platform Management Interface
(IPMI) tool to maintain Lights Out Management (LOM) control of the machines as the servers are Dell 1950 machines.

There are existing scripts to power machines on or off that administrators may use to control the machine states. This does need some exploration to see how useful it is for future iterations of the project. At present, a Fluent interface has been developed around the ipmitool tool, providing a subset of commands to the user. If this needs to be extended, a Python IPMI[1] library may be used.

[1] https://pypi.python.org/pypi/python-ipmi/0.3.0

These scripts are used to temporarily set the machine to boot from the network via Pre Execution Environment (PXE) so that the machine can be installed. The machines are then restarted. When the installation restarts, the machine will boot into the installed operating system. The commands allow machines to be powered on, off or power cycled.

A Redis server[2] is used as a logging engine to store the commands given to the scripts and the nodes that were requested. This provides a simple engine to support the operation and querying of commands that were run.

4.2. Installing the Operating System

A standard Ubuntu 16.04 LTS netboot image with a base user and installation is provided. This is installed over the network, as described in 4.1, so that the installation is repeatable and is known across all the machines managed by that build system.

This image file installs creates a standard user with administration privileges and a minimal installation of the Ubuntu operating system. The networking is created by the DHCP server, DNSMasq. This user is added to the docker group and have an SSH key for the Docker client added to the machine from the control node.

In Figure 4, we show the deployment diagram for the deployment server. We use a server called DNSMasq[3] to control which machines can have the Ubuntu image installed. At present, we only support Ubuntu 16.04 but will add future Long Term Support (LTS) versions of Ubuntu, or the standard operating system for the SKA should this change.

We use it with a controlled list of machines. The MAC hardware address is also used to limit the machines that can be installed as well as linking the machine being installed to a network identity. The IP range served by this build server is limited to the cluster to prevent other machines from trying to install the file.

The server provides TFTP services to support netboot operations. There is a directory within the TFTP location that contains the image and the metadata.

4.3. Preseed Files

We use a Debian preseed file[4] to install the Ubuntu 16.04 LTS operating system in a reproducible manner. Using a standard file that is called by the PXE kernel allows for the:

- clean installation,
- a standard user to be installed with sudo permissions,
- network configuration is automated,
- defined set of services can be installed.

[2] https://redis.io/
[3] http://www.thekelleys.org.uk/dnsmasq/doc.html
[4] https://help.ubuntu.com/lts/installation-guide/armhf/apbs01.html

We note that the installation requires NTP to set up the machines. In this case, we use an institutional Network Time Protocol (NTP) server. An NTP server has not been defined within the build system and it would be beneficial to have one local source defined so that all machines are built using the same time source.

Using the late command section of the file, we can then install software via the script. We need to think about how we update the build server with a notification of installation success. We may also want to store the file and preseed in a database using time and node.

The configuration files are also stored in version control for sustainability issues.

4.4. Installing software stacks

In the above section, we mention the installation of the required software stacks. We use a stored set of scripts that the install different services. These scripts each install one particular service so that the provided infrastructure is kept as minimal as possible.

Unlike other testing services, such as the now archived FutureGrid[5] or Chameleon[6], we do not currently support user contributed images or service scripts. We may need to consider a process for providing this or allowing services to be developed via a feature request.

The preseed file installs the script as the final part of the installation before the machine reboots itself. Future work in this area may be to define the set of the software profiles to be installed and to allow the API to provide these on request.

5. Deployment Server

The second server hosts the Docker registry. This supports the central building of the software containers and a central point of retrieving them. Docker was chosen to reflect the assumptions made in the Software Repository Database Product Tree item. A previous iteration of the system used the Puppet configuration management system.

We present the Docker registry before discussing the pipeline to ingest Docker files.

5.1. Docker Registry

The Deployment server will require a software database, as outlined in figure 3. At its simplest, this may be a Docker Registry instance can be run with an open port using a standard set of commands. It could also be a larger system such as Singularity or Kubernetes, software designed for large scale installations.

[5] http://archive.futuregrid.org/
[6] https://www.chameleoncloud.org/

We have installed self-signed certificate on the master service and the key is installed on the client machines during their installation. Communication between the registry and clients uses TLS by default and limits the clients that can connect to the server to known installations with the correct key.

The registry has a data store that is on the local file system but this can be altered. A piece of prototyping that would be useful is to create two nodes linked by a Network File System (NFS) or an object store such as Ceph. This would provide a back-up system potentially in the case of hardware failure or disaster recovery, where the registry is restored from existing containers rather than having to rebuild them from version control.

An experiment using a Raspberry Pi2 machine was conducted to investigate the effect of low power machines being used as the registry endpoints. At idle, the machines used 2.4 watts of energy and 2.9 watts when a container was being called. Although the project may not use such machines, it does demonstrate that a low power machine approach may keep the deployment systems within the overall power cap.

5.2. Docker ingest pipeline

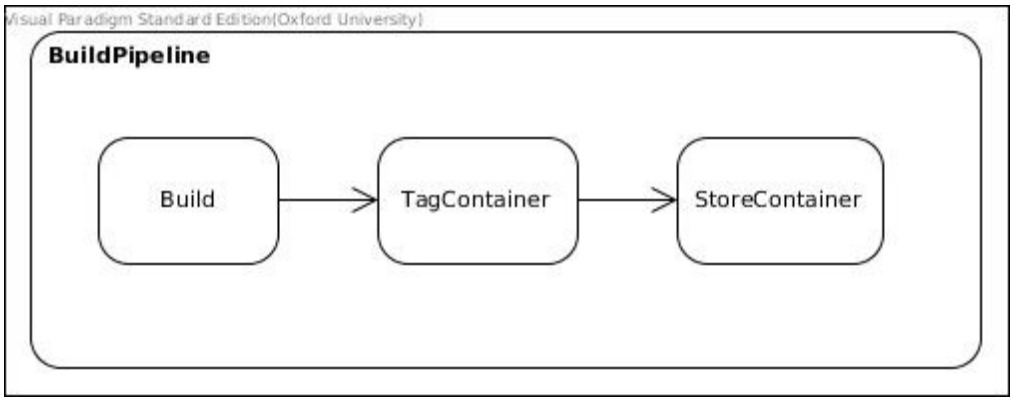

Figure 4: Activity Diagram of Building and Storing a Container

There should be a simple container ingest pipeline, as shown in Figure 4, to build and store the container into the registry that can be automated and run as part of a Continuous Deployment pipeline. This reduces reliance upon on a release team to maintain the container releases and potentially provides the provenance.

Docker build – t <template name > -f <location of Dockerfile> .
Docker tag <tag>:<version>
Docker push <location of registry> / <tag name>

Using the registry on its own without a separate data store allows a test instance to be brought up relatively quickly. Resilience can be provided by giving the registry a location of a data store, which might be a separate disk or potentially an external service.

Using a local registry reduces the reliance upon a third party for access and availability. It also reduces potential latencies across the public Internet.

6. Performance

The nodes are connected by a 1 Gb Ethernet link on a switch that is shared with the Advance Research Computing facility.

The current build time is around 10 minutes to install the image[BM1] and then some time to reboot the machines into a ready state. There are two performance bottle necks that we identify: firstly, the waiting for an NTP service and secondly, the installation of other services via the Preseed command. An unreliable network connection to the package repositories also affects the installation time as the image has to go to the public Internet to download the core packages. Hosting a local mirror may help with this task.

The performance of Docker installations is affected by the underlying storage media. (Sîrbu, 2016) suggests that the netboot image installation would similarly be affected by this as the image has to be read from a disk. The existing Dell 1950 servers both use spindle disks. A future experiment or architecture may investigate using machines with Solid State Disks (SSDs) to gain a performance improvement.

7. Future Work

As part of the future work, we would like to install Open Stack as a separate software profile. This requires work to add in the packages and to also update the API so that the installation profile can be created. It would provide a separate infrastructure for Integration prototyping and support plans for the prototyping platform.

An outstanding issue is the lack of an interface to the LMC to manage the starting and re-installation of the nodes and containers. At the very base model, the interface should provide the IP and the type of node is created. We may desire a node with a particular build, such as Docker or OpenStack. One option is to build a node with all software. A more efficient way of doing this would be to define preseed files with particular builds that are defined by the master controller.

7. 1. Master Controller interface

The Telescope Manager will control the state of the SDP, including re-installing machines. This has not been defined in an Interface Control Document (ICD). We make the assumption that there will be a central build service for the SDP and that all deployments will happen via this service.

There are two issues to address: node management (power on or off) and installation. The service described in 4.1 discusses the existing interface. We support power management and the installation profiles.

7.1.1. Manage

The node management needs two parameters: <node name or IP> <command>. We may limit the command, initially, to power on, off (hard off), or cycle.

7.1.2. Install

The installation of the machine requires the node name or ip and the type of installation, such as Ceph or Docker.

Additional commands, such as the images one discussed below, can be added to the software.

7.1.3. Images

This command will list the stored images and their names.

The installation files use a web interface on the build server to store the files. Echoing the way in which Linux repositories work, the type parameter could refer to a directory. This does allow for the installation to report an error if an incorrect type is requested. An Nginx web server is used to host the files for the build system.

We have not experimented with attaching this to a Tango driver or a RabbitMQ interface, as suggested by Malta. This is would require a simple abstraction to achieve the relevant class interface.

7.4. Deploying instances

When a Dockerfile is run, it has to install a base image. If this is not found within the existing Docker image listing, then it must call this from the main Docker registry. This makes two assumptions: firstly, the Internet connection to the Web is active and not in a degraded state and secondly, the image exists in the remote repository.

It is possible that we need to build a base image that can be held in the local repositories. This mitigates the requirements for the created Docker images to be dependent upon the remote repository. It also allows the project to preserve the image as well for future reference.

This would support software preservation and running operating systems no longer supported by an external service. It does require that the images are built and stored to that they can be shared whilst in operation. This will also support a staging environment to test containers as well as allowing the image to be shared with the relevant teams for their own testing.

A catalogue of existing types should also exist alongside their install profile names for the build API.

MaaS installations have been suggested as supporting emulators for preserving workflows and machine details (Wehrle, 2014). The prototype should maintain a log of what was built at what time and on to what machine to support the science output and results.

The use of continuous deployment of containers into the Software Repository Database alongside using branching in version control would support the automation of the processes. This supports the Integration Prototype's activities and also allows for the testing of DevOps metrics. (Puppet, 2016) identifies two types of metrics: throughput and stability. The former measures the deployment frequency – typically the number of deployments per day – and deployment lead time – the length of time code takes from commit to deployment.

Whilst the SKA is unlikely to deploy to live via continuous deployment, we should measure the time it takes to provide a container within the test deployment system to identify any process bottlenecks, so providing the deployment lead time into the container.

The stability metrics, such as Mean Time to Recovery (MTTR) and change failure rate (CFR), are more useful to the project. MTTR is the amount of time it takes to recover from a failure. This cannot be fully answered by integration prototype build systems but using the MaaS installations should support work in this area. The failure may come from a failed container deployment, a failed container, or an issue with the underlying installation. Part of the performance challenges and testing would be to quantify this for the underlying node installations. The CFR can also begin to be quantified by testing different installations types.

8. Conclusion

A build and deployment infrastructure is described with some of the processes required to install machines repeatedly and reproducibly.

We demonstrate the changing nature of infrastructure deployment technologies from the move from a Puppet system to Docker. Experimenting with integration on to the proposed P3 platform would provide challenges to the deployment system, mirroring those for a heterogeneous, distributed system such as the SKA SDP. This does invite questions about how this might link to the software harmonisation processes to support them and help manage change. In so doing, this improves the understanding of the stability metrics and to plan for them.

We can install the required operating system onto the cluster using a Metal as a Service style deployment to improve the sustainability of the installations and to reduce the chance of failure. We describe how we use a standard configuration to make the installation repeatable and sustainable and identify some performance issues. From this, we identify the requirement for deployment metrics and synergies with the software harmonisation process.

Acknowledgements

The authors acknowledge the FP7 funded CRISP project, grant agreement number 283745, the Square Kilometre Array (SKA) Telescope Project, the Department of Physics and the Oxford e-Research Centre (OeRC) at the University of Oxford, for use of their system resources and personnel. The authors would like to acknowledge the use of the University of Oxford Advanced Research Computing (ARC) facility in carrying out this work. http://dx.doi.org/10.5281/zenodo.22558. The authors also acknowledge the Oxford University CUDA Center of Excellence (CCOE) for providing GPU resources.

References

Desnoyers, Peter, et al. "Hardware as a service-enabling dynamic, user-level bare metal provisioning of pools of data center resources." 2014 IEEE High Performance Extreme Computing Conference (HPEC '14), 2014.

Diaz, Javier, et al. "Abstract image management and universal image registration for cloud and hpc infrastructures." *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*. IEEE, 2012.

Puppet. 2016 state of DevOps report. Retrieved December 1, 2016, from https://puppet.com/resources/white-paper/2016-state-of-devops-report

Sîrbu, Alexandru, et al. "Predicting provisioning and booting times in a Metal-as-a-service system." *Future Generation Computer Systems* (2016).

Wehrle, Dennis, et al. "Emulation-as-a-Service-Workflows and Infrastructure to Support Recomputable Science." *Utility and Cloud Computing (UCC), 2014 IEEE/ACM 7th International Conference on*. IEEE, 2014.