Integration of cloud computing paradigms for performant analysis of simulated process engineering applications

A R&D report (Dissertation) submitted in partial fulfilment

of the requirements for the degree

of

Bachelor of Engineering (Hons)

at

The University of Waikato

by

Caleb Archer

Supervised by:

Tim Walmsley, Mark Apperley

KO TE TANGATA

THE UNIVERSITY OF

WAIKATO

Te Whare Wānanga o Waikato

2024

Abstract

Acknowledgements

Declaration of Authorship

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The process systems engineering (PSE) field is critical to enabling detailed analysis and optimisation of industrial chemical processes. Despite this, modernisation of accessible optimisation software tools and techniques has slowed, with most software-based analysis performed via desktop applications constrained by the resource limits of local systems. More efficient optimisation techniques are available, but require software programming proficiency not held by the common process engineer. The cloud computing sector has significantly advanced the ability to run and deliver software in a distributed fashion, without requiring clients to procure and maintain physical hardware. Cloud-native software systems are better positioned to respond to dynamic usage and allocate computational resources efficiently based on need. There has been minimal convergence of the PSE and cloud computing fields.

This work details and evaluates the development of a cloud-native PSE software platform (the Ahuora Digital Platform) on a physical Kubernetes cluster, comparing the performance of the different software components running on the cluster versus a local deployment of said components.

1.1 Background

1.2 Summary of Literature Review

Chapter 2

Research and Development Approach

2.1 Kubernetes cluster architecture

2.2 Performance analysis experiment structure

2.2.1 Controlled environment

2.2.2 Controlling variables

2.2.3 Preliminary performance modelling

2.2.4 Empirical comparison

2.2.4.1 Grafana k6 load-testing

2.2.4.2 Grafana system monitoring

2.2.5 Key metrics

2.2.5.1 Response time

2.2.5.2 Throughput

2.2.5.3 Quantile comparison

2.3 Performance analysis experiments

2.3.1 Benchmark

2.3.2 Test scenarios

2.3.2.1 Average-load testing

2.3.2.2 Stress testing

2.3.2.3 Soak testing

2.3.2.4 Spike testing

2.3.2.5 Breakpoint testing

2.3.3 Resource allocation

2.3.4 Horizontal scaling policies

Chapter 3

Implementation

3.1 Raspberry Pi hardware and network provisioning

For ease of access and cost minimisation purposes, a set of eight Raspberry Pi 5 computers was obtained to run the Kubernetes cluster on. Each device has an active cooler component installed to effectively cool the CPU (Central Processing Unit) and prevent system throttling. The usage of these devices allows for the construction of a physically compact computing cluster at low cost.

A headless (sans desktop interface) version of Raspberry Pi OS (operating system) was loaded onto eight corresponding SD cards, which each device uses as primary storage. The headless version of the OS strips the resource consumption of the desktop user interface, which is not required, as most interaction with each device will be automated over the network, requiring no more complication than a remote CLI (Command-Line Interface) provides.

To allow the Raspberry Pi devices to communicate and form a computing cluster, a network switch was used. The switch has eight ethernet ports, and each Pi connects to the switch via CAT-6 ethernet cables. On initial start-up, the Pi devices did not have IP addresses assigned, and could only be identified by their MAC addresses, which are not suitable for higher-level communication protocols, which rely on IP addresses as part of the low-level IP protocol (Internet Protocol). Along with this, the devices needed access to the internet, and as such, a network router was required. Because of the network security concerns held by campus ITS (Information Technology Services), it was not appropriate to connect the cluster to the primary university network in order to gain internet access. Instead, a 4G Teltonika RUTX11 cellular router was procured, which could connect to the 2degrees-managed university 4G network, and thereby obtain internet access. With only eight ethernet ports on the switch, cluster nodes (Pi devices) requiring internet access were connected to the router over Wi-Fi. Two of the eight nodes have been configured to access the internet this way.

After this, the nodes still needed IP addresses assigned for the network switch interface. A DHCP (Dynamic Host Configuration Protocol) server called DNSMasq was set up on one of the Wi-Fi connected nodes (called the *ingress* node). DHCP allows for devices on a network to be automatically assigned IP addresses from an available address pool, avoiding the tedious process of manually configuring network interfaces on each individual device. In this context, it was necessary to assign static (unchanging) IP addresses to each device, so DNSMasq was configured to assign specific addresses based on the MAC address of each device. The ingress node was assigned 192.168.100.101, with the rest numbered from 192.168.100.121 to 192.168.100.127. DNSMasq was also used as a DNS (Domain Name Server) server to enable name-based communication over the network, instead of being forced to recall specific IP addresses.

One issue with managing an airgapped (internet-isolated) computer network is the potential for dates and times to become desynchronised, especially following a power outage or any instance where nodes are powered off. Computers typically synchronise with the actual time via time servers accessible over the internet, but perform no such synchronisation without internet access. This problem was encountered during the development process, where logs and metrics that were expected from each node on the cluster were missing. Because of prior cases where some nodes had been powered off for an extended period of time (several weeks), and a lack of hardware clocks on Raspberry Pi models, the date and time on these nodes lagged by more than a month. This caused the logs and metrics to be rejected by the monitoring tool to which they are uploaded. To resolve this, an NTP (Network Time Protocol) server (ntpsec) was deployed on the ingress node, and all airgapped nodes were configured to use the ingress node as their time server. Since the ingress node can retrieve the actual time over the internet, it can provide the actual time to its airgapped clients.

3.2 Ansible playbook automation

To manage the bulk of device and system-level configuration of the cluster, a configuration automation tool called Ansible was heavily used throughout the development process. Ansible allows a developer to define a *playbook*, which consists of a set of tasks that will be run in sequence on a target host (a remote device), or a group of hosts. These tasks may be defined as arbitrary shell commands, but they often provide a higher level of abstraction, where a developer can easily specify the parameters from a restricted set, rather than having to remember an exact series of (potentially confusing) mnemomics.

Playbooks can also include *roles*, which include their own sets of tasks, but are focused on grouping related tasks together, and allowing parameters to be easily re-used amongst said tasks. If made analogous to an imperatively written programming language, a playbook is like a program, and a role is akin to a class or module.

The Ansible playbooks used in configuring the system were adapted from the official K3s Ansible repository[1], with many modifications. Included playbooks are: *reboot*, for restarting all Kubernetes cluster nodes; *registry*, for setting up a container image registry mirror; *reset*, for removing configuration and components installed via the site playbook; *site*, for performing the entire cluster software installation and configuration process; and *upgrade*, for updating installed cluster software to a new version.

- *airgap*: Configures hosts for an air-gapped environment.
- *k3s_server*: Configures K3s control (master) nodes.
- *k3s_agent*: Configures K3s agents (worker nodes).
- *k3s_deployments*: Configures third-party software to be deployed at installation time.
- *k3s_upgrade*: Performs K3s upgrading process.
- *ntp_time_server*: Installs an NTP time server and configures clients to use it.
- *prereq*: Performs any prerequisite configuration before cluster start-up.
- *raspberrypi*: Performs configuration on all remote hosts specific to Raspberry Pi systems.

3.3 Isolated cluster access

3.3.1 Network airgapping

As a step towards better cluster security, access to the internet for nodes running Kubernetes is heavily restricted. No nodes beyond the control node are connected to the 4G router, and are limited to local network communication. In the case of maliciously crafted or modified software that may inadvertently be deployed on the cluster (or directly to the base hosts), their ability to exfiltrate information or otherwise communicate with the outside world has been minimised. The K3s control node requires internet access in order to perform tasks such as push collected logs and metrics, check for configuration updates in the associated cluster GitHub repository and install plugins.

[1]K3s Ansible GitHub repository: https://github.com/k3s-io/k3s-ansible

3.3.2 ZeroTier Virtual Private Network (VPN) usage

With these network limitations in place, it was still necessary to have a method to remotely access and manage nodes within the cluster, while maintaining security requirements. While it was possible to log in to the cluster via an authenticated laptop connected to the cluster 4G router, this was certainly the least convenient option, especially when working from a remote location, or even an office desktop using a separate network. On the other hand, exposing an SSH (Secure Shell) server port to the outside world would have provided convenience, but less security confidence.

ZeroTier[1] is a VPN (Virtual Private Network) service that allows users to connect devices of varying types to a virtual network that appears to behave the same way as a physical network switch. IP addresses from private subnets are assigned dynamically to each authorised device, which can then communicate with other devices on the network from anywhere in the world, as long as the device has internet access. ZeroTier provides a set of free root nodes that facilitate the establishment of connections between devices, which then continue to transmit data over a direct P2P (peer-to-peer) connection. Traffic between devices is end-to-end encrypted, meaning that in-flight data cannot be intercepted and interpreted by actors in the middle of a connection, even ZeroTier themselves (when a P2P connection cannot be established and has to be relayed via ZeroTier nodes).

The cluster ingress node is the core device connected to a ZeroTier VPN. Other devices that need to remotely access the cluster do so first by joining the VPN, and then creating an SSH connection to the ingress node via the ingress node's VPN-allocated IP address. Following this, the rest of the physical cluster network can be accessed over SSH via the ingress node.

With this strategy, any services that need to be exposed to developers but not to the wider internet can be utilised via the VPN, providing a simple, secure and convenient management context. During early development of the Ahuora Digital Twin platform, there was no authentication system in place, but stakeholders of the platform needed to be able to test it without having to set up a manual deployment. To enable testing, the control node of the cluster was added to the VPN, and then the device to perform the testing from, where it could then access the front-end of the platform securely.

[1]ZeroTier VPN: https://www.zerotier.com/

3.3.3 Private container image registry mirror

Because of the internet access restrictions on the cluster, Kubernetes Pods attempting to retrieve container images from external sources (such as Docker Hub) will repeatedly fail to deploy. This presents a problem, as some form of external access is required, but it is not acceptable to provide broad internet access to all cluster nodes. In this scenario, it is necessary to use some form of limited proxying solution, where requests can be made to a service local to the network, which has access to the internet, and can pull container images from external image registries on behalf of clients.

A container image mirror service called *oci-registry* was used to achieve this functionality. The ingress node was configured to run oci-registry and expose it to the physical network. All nodes within the Kubernetes cluster are likewise configured to make image pull requests to the ingress node. When a request is made, oci-registry[1] checks if the requested image is present in its cache. If it is not present, it retrieves it from the requested source; otherwise, it is served from the cache.

[1]Container image registry: https://github.com/mcronce/oci-registry

3.3.4 Cloudflare Tunnel ingress point

3.4 Kubernetes manifest configuration

3.4.1 Deployments

3.4.2 Services

3.4.3 Ingresses

3.4.4 Authentication

3.4.5 Database Stateful Set

3.4.5.1 Postgres

3.4.5.2 Connection Pooling

3.5 Kubernetes deployment automation

3.5.1 ArgoCD manifest synchronisation

3.5.2 GitHub Actions platform release pipeline

Chapter 4

Results and Discussion

4.1 Outcomes

4.1.1 Resource allocation

4.1.2 Horizontal scaling policies

4.1.2.1 Minimum replicas

4.1.2.2 Target resource utilisation

4.1.2.3 Variable load conditions (stabilisation window)

4.1.3 Clustered vs. local PSE optimisation performance

4.2 Impacts

Efficiency, accuracy, feasibility (examples of demonstrable impacts).

4.3 Threats to validity

Chapter 5

Conclusion and Future Work

References