Experimental Plans for MicroK8s and NERC OpenShift

Jonathan Chamberlain

February 2024

1 Introduction

This document is a detail of the research plan to simulate attacks on Kubernetes auto-scaling by myself and Jilin. We plan to execute this both locally on NISLAB equipment to effectively serve as proof of concept, and on our NERC OpenShift instance in order to provide demonstration on a more *realistic* platform.

2 Tools

Canonical MicroK8s - `https://microk8s.io/`

MicroK8s is an open-source, light-weight production Kubernetes designed to be deployed in low resource environments, supporting developments of Kubernetes in unusual environments or proofs of concept on smaller scales before introducing to larger clusters. MicroK8s is scalable from a single node to high-availability production cluster, and includes Prometheus as part of its core add-ons for monitoring. Currently we have MicroK8s installed on a pair of Lenovo M Series ThinkCentre desktops in the NISLAB workspace to serve as proof of concept testing in parallel with experiments on other platforms, as well as ensure that we have at least one experimental platform entirely under our control for any future concept testing. This also enables us to do the bulk of our testing locally, and thus save on any charges assessed for Service Units expended, optimizing our OpenShift experimental runs.

Red Hat Open Shift on NERC - `https://www.redhat.com/en/technologies/cloud-computing/openshift`, `https://nerc.mghpcc.org/`

Open Shift is an open source Kubernetes-powered container management platform, which comes in both managed and self-managed varieties deploy-able in a range of environments, including commercial providers such as Amazon Web Services and MS Azure. The primary selling point of Open Shift is to simplify deployment of cloud applications. As Open Shift is in wide use we believe that experiments on this platform can be considered representative of auto-scaling

behavior in production environments. As we use OpenShift as deployed to the New England Research Cloud (NERC), the version of OpenShift we use in our experiments is the Open Shift Container Platform, which is unmanaged. The main advantage from our POV however is that this enabled additional customization over the configuration settings.

In particular, OpenShift supports Advanced Cluster Management to enable autoscaling at both the node level for scaling the numbers of containers/replicas, and has built in monitoring support based on Prometheus (and uses Prometheus components in addition to components defined specifically for OpenShift).

Apache jMeter - `https://jmeter.apache.org/`

jMeter is an oepn source tool utilized for load testing; among its features includes the ability to support shell scripts, HTTP(S), FTP, Mail protocols, etc., as well as multi-threading for concurrent sampling. This yields maximum flexibility over the types of payloads we can experiment with should we determine later that the model proposed below is insufficient and we require a different server operation, or more complex server operations. The scripting ability allows for variable input, and thus we can vary the arrivals according to statistical distributions to prevent deterministic based results for the customers.

Prometheus - `https://prometheus.io/`

Prometheus is an Open-Source monitoring and alerting tool which scrapes time-series data from cloud environments. The data can then be queried utilizing the PromQL query language. These queries can be leveraged to send out alerts, or enable custom metrics to be generated for cluster management (e.g. autoscaling criteria). Prometheus is a standard open source tool, to the point where Prometheus based monitoring is shipped as a standard add-on or directly integrated into Kubernetes management platforms as seen above.

Grafana - `https://grafana.com/`

Grafana is a visualization tool which takes PromQL queries and plots the results returned. Dashboards are customizable to display the results of specific queries and/or target pods or containers.

MockFtpServer - `https://mockftpserver.org/`, `https://github.com/dx42/MockFtpServer`

MockFtpServer is a tool which provides mock/dummy FTP servers which provide the means to test basic FTP functions, including logging in, file retrieval, sending files, and listing the directory.

3 Assumptions

The key assumptions are that we have a known Poission (exponential) distribution of customer arrivals implementable in jMeter load testing, and that users

accessing the service are only ever downloading files (we can imagine that the use case we are modelling is that of freeware distribution, a la Linux, although the actual payloads will be much smaller due to file size limitations). In addition

- We assume that we are able to scrape Prometheus to grab custom metrics for auto-scaling.
- We assume that we are able to scrape Prometheus to compute the total system delay of requests in order to make comparisons.
- We assume that we have a means to track dropped requests, whether through Prometheus or jMeter.
- We assume that experiments can be configured to be run indefinitely, and thus suspended only we are satisfied we have sufficent data, or otherwise pre-configured to run over an extended period of time.
- We assume we are able to configure the auto-scaling to configure identical FTP servers to accommodate file downloads (given the use case modeled, this should be a reasonable assumption given that files are being offered).
- We assume that the FTP server is configured to offer a series of static file downloads of variable size.
- We assume that cluster autoscaling only enables a single node to be added or removed at a time (this is based on the cluster auto-scaling provisions)
- We assume that the above notwithstanding, pod autoscaling is customizeable in terms of rate of scaling and delay before scaling.
- We assume that the attacker has a means to identify the state of the system before launching an attack (for the applicable experiments).
- We assume that each pod consumes one standard service unit (1 vCPU, 4 GB vRAM), with associated cost 0.013 per compute hour.

4 Constant Load Experiments

The purpose of the constant load experiments are to provide sanity checks that the experimental platforms and our tools are working as expected. These results are not intended to be included as part of our published results, but rather provide confidence in the correctness of the results of further experiments due to the controlled nature of the traffic sent.

For the initial wave of experiments in this section, default values will be utilized for aspects not being tested in order to do control testing and to identify candidates for which aspects to combine for optimality. The load will consist of sending requests every 0.5 seconds to download a file from the FTP server. We assume a maximum cluster size of 30 pods. In a multi cluster setting we assume we have up to 4 total clusters (as cluster autoscaling is assumed to have +/- 1 cluster at each step, we focus on the HPA level settings in our test plans).

Target tests

- CPU Utilization target: 50%[1], 75%, 100%
- Memory Utilization target: 50%, 75%, 100%
- Request target (measured as bytes received per minute per PromQL query metrics): 10, 50, 100

Upscale testing

- Stabilization Window (delay): 0 sec, 60 sec, 120 sec
- Scaling percentage: 50%, 100%, 200%
- Percentage Scaling period: 15 sec, 30 sec, 60 sec
- Scaling by number: 1, 2, 4, 8
- Number Scaling period: 15 sec, 30 sec, 60 sec

Downscale testing

- Stabilization Window (delay): 60 sec, 120 sec, 180 sec, 300 sec
- Scaling percentage: 50%, 100%, 200%
- Percentage Scaling period: 15 sec, 30 sec, 60 sec
- Scaling by number: 1, 2, 4, 8
- Number Scaling period: 15 sec, 30 sec, 60 sec

Initial testing: run loads for 10 minutes on the local MicroK8s set up testing each of the above settings and that scaling occurs with respect to the setting selected.

Advanced testing: pick combinations of up and down scaling settings to judge differences in behavior (e.g. choosing to combine scaling by number and scaling percentage policies for more rapid up-scaling/limit up-scaling beyond a certain cap or mixing and matching stabilization windows). Alternate periods of sending traffic and not sending traffic to test responses of re-scaling following idle periods.

[1]This is the setting used in tutorial examples and therefore can be used as the baseline control for all other testing scenarios.

5 Variable Load Experiments

The specific parameters will depend in part on what has been identified as the aspects to key in on in the previous section, and will be updated as appropriate. However, what has been identified:

- The test FTP server is to be fleshed out with additional files of various sizes, with the test loads randomized not only in inter-arrival periods, but in the files requested for download.
- The tests will focus primarily on testing the target metrics to scale on, with the stabilization windows and scaling periods/percentages/limits as a secondary concern and set based on results from the Constant Load tests.
- Tests will run over a longer (6-24 hour) period to achieve a wider sample of behavior.
- Traffic rates are to be derived from production cloud trace data.

6 Attacker Yo-Yo Load Experiments

The "legitimate" customer traffic shall be similar to that in the previous section in that it is variable load conducting various tasks. As the attacker's goal is to waste resources, the attacker traffic consists of spamming requests for the file structure of the server at a high rate to accomplish the goal of wasted work which accomplishes nothing useful for the victim. Otherwise, the experiment plan is similar to that of the previous section, with the exception of sending waves of requests for file information for 120 seconds in 0.1 second bursts, every 10 minutes for local testing; and for 10 minute bursts every 30 minutes on the NERC cluster.

7 Attacker Dynamic Load

TBC - a key underpinning assumption is that the attacker has an idea of the state of the system; further reading on how to determine this is necessary to flesh out the plan.