ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-TH2A002

HIGH AVAILABILITY ALARM SYSTEM DEPLOYED WITH KUBERNETES

J. Bellister, T. Schwander, T. Summers
SLAC National Accelerator Laboratory, Menlo Park, USA

Abstract

To support multiple scientific facilities at SLAC, a modern alarm system designed for availability, integrability, and extensibility is required. The new alarm system deployed at SLAC fulfils these requirements by blending the Phoebus alarm server with existing open-source technologies for deployment, management, and visualization.

To deliver a high-availability deployment, Kubernetes was chosen for orchestration of the system. By deploying all parts of the system as containers with Kubernetes, each component becomes robust to failures, self-healing, and readily recoverable.

Well-supported Kubernetes Operators were selected to manage Kafka and Elasticsearch in accordance with current best practices, using high-level declarative deployment files to shift deployment details into the software itself and facilitate nearly seamless future upgrades. An automated process based on git-sync allows for automated restarts of the alarm server when configuration files change eliminating the need for sysadmin intervention.

To encourage increased accelerator operator engagement, multiple interfaces are provided for interacting with alarms. Grafana dashboards offer a user-friendly way to build displays with minimal code, while a custom Python client allows for direct consumption from the Kafka message queue and access to any information logged by the system.

INTRODUCTION

To assist in the commissioning of the upgrade to the Linac Coherent Light Source (LCLS-II) at SLAC, an upgrade to the alarm system was proposed. In performing this upgrade there were several main priorities. First is a system which would be easy to work with for end users. Both adding new devices to be monitored, as well as interacting with alarms that are surfaced should be as straightforward as possible. We also wanted a system that had near constant uptime, as it would be monitoring multiple critical facilities across the lab. Finally, it was important to adhere to current best practices around alarm management and system deployment as much as possible. To that end we assessed the current status of various alarm management systems as well as deployment options.

The Phoebus alarm server [1] met most of our requirements so we chose to base our system on it, thus eliminating the need to reinvent the wheel for basic alarm logic and functionality. For the deployment process Kubernetes [2] was settled on for orchestration of the system. Using these technologies brings our system in line with current best practices, while meeting our goal of high availability and performance. Further details of the deployment process follow.

ALARM SERVER

Since the Phoebus alarm server meets most of our requirements, only a few site-specific changes were made to it prior to deployment. Two modifications of note are to allow the same process variable (PV) to appear along multiple branches of the alarm tree, and a way to make it easier for an alarm server to reload when its underlying configuration file has changed. This allows for an automated continuous deployment process in which user changes to alarm configurations can be automatically incorporated into running alarm servers.

The main functionalities of handling alarm logic and translating changes in EPICS alarm severities into Kafka [3] messages remain unchanged. As depicted in Fig. 1 below, the alarm server monitors PVs from an EPICS source, in this case a gateway. Any relevant changes are translated to Kafka messages and stored in the proper alarm topic. All clients to the system read from the Kafka queue.

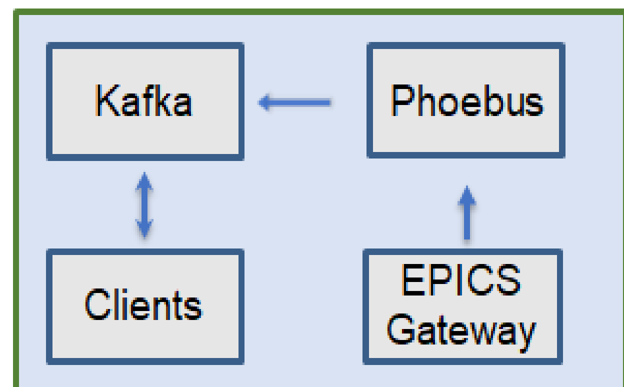

Figure 1: Simplified flow diagram of alarm data.

ALARM LOGGER

The other main functional component of the system is the Phoebus alarm logger. This records a history of alarms and actions taken on them, and persists this history into an Elasticsearch [4] data store.

Our deployment also provides the ability to use Loki [5] as a data store instead of Elasticsearch. This is handled by using a simple python application that reads data from Kafka and pushes it to Loki. Since many systems at SLAC already use Loki for log storage, providing this flexibility allows users the choice of where to persist data, and more easily integrate it with the data storage of other applications.

Content from this work may be used under the terms of the CC BY 4.0 licence (© 2023). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-TH2A002

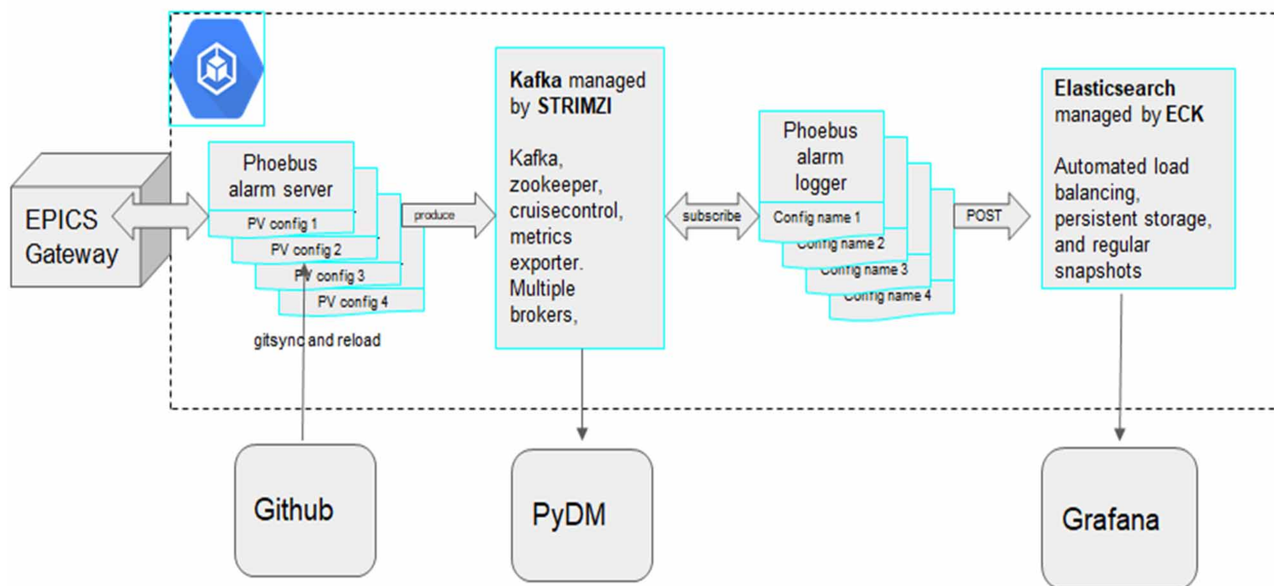

Figure 2: Diagram of the components deployed into the Kubernetes cluster. Multiple replicas of each component are running, while clients outside the cluster are able to access the data they need.

KUBERNETES DEPLOYMENT

To orchestrate the containerized deployment, Kubernetes was chosen to fulfill all requirements of the deployment while keeping up with current best practices. A diagram showing the deployment appears above in Fig. 2.

Containerization

As we have some site-specific changes to the Phoebus components, the containerization of those is done by creating our own Docker files and associated options for application start up. These are available at our GitHub repository for the container images. For Kafka and Elasticsearch, Kubernetes operators are used for the deployment and described in further detail below.

Kubernetes Cluster

In order to proceed with a Kubernetes based deployment, a Kubernetes cluster must first be available. At SLAC, we have an on-premises data facility in which we can run Kubernetes. This is called the SLAC Shared Scientific Data Facility (S3DF) [6].

To facilitate deployments of multiple applications across different user bases at SLAC, virtual clusters (vclusters) are used. Similar to virtual machines, these vclusters allow for multiple functional Kubernetes clusters to be run on the same host cluster, without degradation of the functionality provided to the end user.

High Availability

Kubernetes provides many high availability features to ensure we get as close to constant alarm system uptime as possible. If a pod fails, Kubernetes will ensure that a replacement is ready to take over from the failed pod without any disruption to end users of the application. And a user-specified liveness probe can be set to detect problems in the application. This ensures that the potential failures being monitored by Kubernetes match what the user is expecting.

For our deployment at SLAC, we have ensured that there are multiple replicas running of all critical components of the system. This includes a second copy of the Phoebus alarm server for each alarm topic being monitored. While this does result in duplicate Kafka message being generated, de-duplication is handled gracefully on the client side. The benefit of this set up is that any potential downtime is brought as close to zero as possible. Since there is always a backup copy of each alarm server running, the failure of a single server should go unnoticed by the end user.

To further defend against potential downtime, a couple of additional safeguards have been put in place. The option to set pod anti-affinity has been used for the alarm servers, meaning that when choosing which nodes each alarm server should run on, Kubernetes will attempt to ensure that each alarm server and its backup will be running on different nodes. As an example, let's say we have two copies of the alarm server monitoring the “accelerator” topic. When the pods are first scheduled to run, they will be placed onto different nodes. Then if one happens to go down due to a hardware failure on the machine it is running on, the other will still be active and serving data it is on a separate node.

Another option in play is the pod disruption budget. By setting this value, we specify that there must always be at least one instance of each alarm server running at any given time. In the event that servers must be migrated to new pods, this prevents both from being stopped at the same time.

Configuration

To manage the overall deployment, Kustomize [7] is used as the configuration management tool. This provides

Content from this work may be used under the terms of the CC BY 4.0 licence (© 2023). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-TH2A002

Content from this work may be used under the terms of the CC BY 4.0 licence (© 2023). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI

a declarative way to configure every component within the deployment, while being fully integrated with the kubectl [8] command line tool. All manifest files for the deployment are version controlled ensuring that it is easy to see why various changes were made and what exactly is running at any given time.

Operators

Operators are a Kubernetes deployment pattern that attempt to automate repeatable tasks that would usually be handled by system operators. They can make managing and updating components deployed to the cluster both easier and faster. For this deployment we have chosen two popular and actively maintained operators for managing the Kafka and Elasticsearch deployments.

Strimzi [9] was chosen as the operator for Kafka, and ECK [10] for Elasticsearch. Both provide tools for helping manage initial deployments and future upgrades.

UPDATES TO THE DEPLOYMENT

There are two main categories when it comes to updates to the running system: users wanting to add a new alarm configuration or modify an existing one, and administrators of the Kubernetes vcluster wanting to update one or more running components. For each case, we have attempted to make the upgrade process both simple to do and robust to errors.

Alarm Configuration Updates

When a user is ready to deploy a new device that will produce new PVs, they may want to add these PVs to an existing alarm configuration file so that they will be monitored by the alarm system. In order to make this process as easy as possible for the end user, nearly all of it has been automated. The user only needs to edit a csv file, at which point a series of GitHub actions take over to validate the change, generate the associated xml file, and lint that file. Once the change has been approved it will be merged into the GitHub repository.

To continue this automatic process, a Kubernetes sidecar has been set up within each pod that holds an instance of the Phoebus alarm server. A sidecar is a deployment pattern in which the sidecar container enhances the functionality of the main container by running alongside it in the same pod. In this case, our sidecar is using git-sync to monitor changes to the GitHub repository. When a change is detected, it pulls in the xml file for its topic and compares it to the most recent one it loaded. If there is a difference, it will send a restart message to the command topic in Kafka using kcat, causing the alarm server to load the new configuration.

Kubernetes Cluster Updates

The second main update scenario is when it is time to move to a new version of one of the components running within the cluster. In the case of one of the Phoebus components, the source code is updated to the latest tagged release, and the Docker image is rebuilt and tagged as a new release before being published to Docker Hub. All manifest files referring to the updated Phoebus component just need to have their release number updated, at which point they can be redeployed.

When updating Kafka or Elasticsearch, we can use the updated versions of their respective operators. To test updates prior to releasing them to production, multiple Kubernetes namespaces can be used. A namespace provides a way of isolating groups of resources within the same cluster. Thus, it is possible to have a staging namespace where changes can be tested and proven to work before then releasing to the production namespace.

USER INTERFACES

To facilitate increased engagement with alarms, multiple additional interfaces to the system have been provided.

Python Client

The first is a python client with the same general functionality as the existing ALH and Phoebus clients. It includes alarms in both a tree and table view, along with the option to take actions such as acknowledge and bypass. Extra functionality provided by this client includes the ability to integrate with the Python Display Manager (PyDM) through use of python entry points. When the python client is installed into the same virtual environment as PyDM, the user will be able to use a data plugin to communicate with Kafka via a PyDM display in the same way as any other data source. This allows for building displays that show the status of top level summary alarms without the need for a separate alarm IOC backing them.

Slack Integration

Another point of integration is with our Slack workspace. Channels can be created for specific alarm topics and users who want to monitor those topics are added to them. To send alarms to the Slack channel, the Slack API is used via Slack Apps support. A webhook is added to the channel to send alarms to, and an App is created and connected with that webhook for posting the messages to the channel. Then a simple container is created within the Kubernetes cluster for reading messages from the Kafka queue for its assigned topic and posting them to the Slack channel. Like the other components of the cluster, this container will automatically restart on any failure minimizing any potential downtime.

Grafana

Lastly Grafana provides a simple way for interacting with not just currently active alarms, but the entire history of both alarm status and actions taken on them. Some default dashboards have been created for users to monitor the history of alarms, and Grafana also makes it easy for users to write their own queries which can eventually also be made into custom dashboards. Since Grafana supports both Elasticsearch and Loki as data sources, we get this support with minimal setup required.

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-TH2A002

CONCLUSION

This upgraded system has been running and monitoring devices in our experimental hutches and cryoplant for several months now. User feedback has been positive, both for the process of adding new devices to be monitored, as well as the options for interfacing with the system. As a result of this good feedback, this type of Kubernetes based deployment is expected to be applied to additional applications in the future at SLAC.

REFERENCES

[1] Phoebus Alarm Server, https://github.com/ControlSystemStudio/phoebus/tree/master/app/alarm

[2] Kubernetes, https://kubernetes.io

[3] Kafka, https://kafka.apache.io

[4] Elasticsearch, https://www.elastic.co/elasticsearch/

[5] Loki, https://grafana.com/oss/loki

[6] S3DF, https://s3df.slac.stanford.edu/public/doc

[7] Kustomize, https://kustomize.io

[8] kubectl, https://kubernetes.io/docs/reference/kubectl

[9] Strimzi, https://strimzi.io

[10] ECK, https://github.com/elastic/cloud-on-k8s

Content from this work may be used under the terms of the CC BY 4.0 licence (© 2023). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI