ISBN: 978-3-95450-209-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2019-WESH2001

CS-STUDIO ALARM SYSTEM BASED ON KAFKA*

K.-U. Kasemir, Oak Ridge National Laboratory, Oak Ridge, USA

Abstract

The CS-Studio alarm system was originally based on a relational database and the Apache ActiveMQ message service. The former was necessary to store configuration and state, while the latter communicated state updates and user actions. In a recent update, the combination of relational database and ActiveMQ has been replaced by Apache Kafka. We present how this simplified the implementation while at the same time improving performance.

OVERVIEW

The Control-System-Studio (CS-Studio) alarm system was developed to support control room operators [1]. It monitors a configurable list of process variables (PVs) from the Experimental Physics and Industrial Control System (EPICS), using either the Channel Access or the newer PV Access network protocol [2]. Whenever a PV enters an alarm state, the alarm system remembers the time and value of the alarm. It lists active alarms on a user interface and will optionally also verbally annunciate each new alarm. Operators can inspect the list of active alarms, access guidance on how to handle the alarm and open display panels that offer more detail for the affected subsystem. After operators acknowledge the alarm and the PV returns into a normal state, the alarm clears.

The alarm system is fully distributed. You may install one or more alarm configurations, and one or more operators can concurrently interact with them.

ORIGINAL IMPLEMENTATION: RDB & JMS

The alarm system needs some way to store its configuration, consisting of the PVs to monitor, their associated guidance and related display links. In addition, the system requires a message bus to communicate PV state changes, annunciation messages and operator actions like acknowledgements.

In the original implementation, a relational database (RDB) stores the configuration, and Apache ActiveMQ provides the message bus (JMS). Each alarm configuration is handled by one alarm server process, typically running as a Linux service. Operators can start one or more alarm clients to interact with the alarm system. All servers and clients can share the same RDB and JMS instance.

This appeared to be a good design choice at the time because an RDB is very well suited for storing information, while JMS is a high-performance message bus. The implementation of the alarm system toolkit was, however, complicated by the fact that interactions with storage and the message bus typically overlap.

When an alarm client starts up, it reads the configuration, which can take several seconds. While it is reading one such snapshot of the configuration, other alarm clients may concurrently change the configuration. Such changes are communicated via the message bus. A proper alarm client implementation thus needs to:

1) Subscribe to JMS and buffer received change indications in memory.
2) Read the last saved configuration from the RDB.
3) Apply the buffered changes to update the in-memory configuration to the current state.
4) From then on directly apply received JMS change indications to the in-memory state.

Similarly, the alarm server implementation needs to:

1) Send alarm state changes via JMS so that running alarm clients can receive them with minimal delay.
2) Write the most recent state of each alarm to the RDB, so that new alarm client instances can start with the most recent alarm state as persisted in the RDB and don't need to wait for a state change via JMS to be up-to-date.

While this was successfully implemented, details of the software appeared overly complicated. Of the two key support technologies, JMS and the RDB, the latter was a performance bottleneck. Sending all messages through JMS on the other hand proved to be very efficient and allowed the addition of logging and analysis tools, for example to determine which alarm triggers most frequently, without impacting the running alarm servers and clients [3].

NEW IMPLEMENTATION: APACHE KAFKA

When porting the alarm system toolkit from the original Eclipse-based CS-Studio development to a standalone platform [4], we used this opportunity to investigate new technologies for storage and message bus, i.e. the RDB and JMS functionality.

* This manuscript has been authored by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the US Department of Energy (DOE). The US government retains and the publisher, by accepting the article for publication, acknowledges that the US government retains a nonexclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for US government purposes. DOE will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

Apache Kafka

Apache Kafka is a messaging service similar to JMS. Multiple clients can connect to the Kafka service and exchange messages. Kafka can be deployed as a single server or a distributed, load-balancing cluster. Kafka supports multiple ways of "streaming" messages. For example, it can store messages and then very efficiently send all stored messages to each newly connected client. Moreover, it supports "Log Compaction", which is very helpful for storing the alarm system configuration and state [5]. By using Log Compaction, Kafka was able to not only handle the JMS functionality of communicating changes, but also the RDB functionality of storing the alarm configuration and state.

The alarm configuration is hierarchical. For example, all alarms related to the accelerator vacuum can be placed in a "Vac" section below a top level "Accel" configuration. Guidance messages and links to related displays apply to all sub-sections of the hierarchical configuration, which often simplifies the task of configuring the system because, for example, contact information that applies to all vacuum alarms can simply be attached to "/Accel/Vac" and there is no need to manually assign it to each vacuum related PV.

Kafka messages consist of a key and a value. We use the path to an alarm configuration item as the message key, and the message value holds the configuration detail. For example, when a user adds two PVs to the alarm configuration, then updates the description of the first PV, and finally adds some Vacuum guidance, the Kafka message stream looks like this, ordered by time starting with the oldest message:

config:/Accel/Vac/PIn = {"description":"Pressure"}
config:/Accel/Vac/POut = {"description":"Outlet Pres." }
config:/Accel/Vac/PIn = {"description":"Inlet Pressure"}
config:/Accel/Vac = {"guidance":"Call 123-456-8910"}

Similarly, when the alarm server detects an alarm on a PV, then on another PV, an operator acknowledges the first alarm and finally the alarm on the first PV clears, the alarm server sends the following state update messages:

state:/Accel/Vac/PIn = {"severity":"MINOR"}
state:/Accel/Vac/POut = {"severity":"MAJOR" }
state:/Accel/Vac/PIn = {"severity":"MINOR_ACK"}
state:/Accel/Vac/PIn = {"severity":"OK"}

Actual alarm system messages contain additional information, for example time stamps, which are omitted for clarity.

Kafka Log Compaction

Over time, PVs change their state and users update the alarm system configuration. Kafka offers several options for handling the resulting growing stream of Kafka messages.

Kafka can be configured to only distribute messages to listeners without persisting them. We use this mode for "command" messages, sent by the user interface when the operator acknowledges an alarm. The alarm server receives the acknowledgment, updates the alarm state, and the command is then no longer needed.

Alternatively, Kafka can be configured to store all received messages. When a client connects to Kafka, it can receive a replay of all past messages. This is useful to analyze the history of alarms, for example to detect the most frequent alarm. In an operational setup, however, this is not practical. The persisted message stream will eventually exhaust the available disk space. More important, an operator who opens the alarm client is not interested in a time-consuming replay of obsolete messages. He/she needs to know the current state of the alarm system.

Kafka can automatically delete older messages after a configurable time, which avoids disk space limitations, but is not useful for the alarm system configuration and state messages. For example, a "config" message for the guidance of accelerator vacuum alarms may be 2 years old, but there was never another "config" message for the item because the same guidance remains applicable. The state of a specific PV on the other hand may change every second, generating numerous "state" messages, and all but the very last one are obsolete. Simply deleting all messages older than, for example, a day would thus drop valid configuration data yet keep obsolete state messages.

Kafka "Log Compaction" is a retention mode that turns out to be perfectly suited to storing alarm system config and state messages. With Log Compaction, Kafka periodically transitions messages from the active segment where new messages are added to a long-term storage segment. In the long-term storage segment, all but the last message for each key are deleted.

To illustrate, assume we generated the previously shown messages in the following interleaved order, i.e. users changed the configuration while state updates occurred:

<u>Active Segment:</u>
config:/Accel/Vac/PIn = {"description":"Pressure"}
config:/Accel/Vac/POut = {"description":"Outlet Pres." }
state:/Accel/Vac/PIn = {"severity":"MINOR"}
state:/Accel/Vac/POut = {"severity":"MAJOR" }
state:/Accel/Vac/PIn = {"severity":"MINOR_ACK"}
config:/Accel/Vac/PIn = {"description":"Inlet Pressure"}
config:/Accel/Vac = {"guidance":"Call 123-456-8910"}
state:/Accel/Vac/PIn = {"severity":"OK"}

Now let's assume that the first six messages are old enough to move into the long-term segment:

<u>Long-Term Segment:</u>
config:/Accel/Vac/PIn = {"description":"Pressure"}
config:/Accel/Vac/POut = {"description":"Outlet Pres." }
state:/Accel/Vac/PIn = {"severity":"MINOR"}
state:/Accel/Vac/POut = {"severity":"MAJOR" }
state:/Accel/Vac/PIn = {"severity":"MINOR_ACK"}
config:/Accel/Vac/PIn = {"description":"Inlet Pressure"}

<u>Active Segment:</u>
config:/Accel/Vac = {"guidance":"Call 123-456-8910"}
state:/Accel/Vac/PIn = {"severity":"OK"}

Messages in the long-term segment are then compacted by keeping only the last message for each key. Previous messages for the same key are removed from the long-term segment. Applied to the above example messages, the long-term segment will be left with only the following messages.

Long-Term Segment:
config:/Accel/Vac/POut = {“description”:”Outlet Pres.” }
state:/Accel/Vac/POut = {“severity”:”MAJOR” }
state:/Accel/Vac/PIn = {“severity”:”MINOR_ACK”}
config:/Accel/Vac/PIn = {“description”:”Inlet Pressure”}

Active Segment:
config:/Accel/Vac = {“guidance”:”Call 123-456-8910”}
state:/Accel/Vac/PIn = {“severity”:”OK”}

Kafka transparently performs this message stream segmentation and Log Compaction without directly exposing details to clients. When an alarm system tool connects to Kafka, it simply receives a stream of messages based on the remaining long-term segment items and the active segment.

config:/Accel/Vac/POut = {“description”:”Outlet Pres.” }
state:/Accel/Vac/POut = {“severity”:”MAJOR” }
state:/Accel/Vac/PIn = {“severity”:”MINOR_ACK”}
config:/Accel/Vac/PIn = {“description”:”Inlet Pressure”}
config:/Accel/Vac = {“guidance”:”Call 123-456-8910”}
state:/Accel/Vac/PIn = {“severity”:”OK”}

In this example, the stream contains only the relevant “config” messages. The “state” messages include two updates for the “PIn” PV, we receive both the older “MINOR_ACK” and the final “OK” state.

Log compaction thus guarantees that the order of messages is preserved. For each key, i.e. for each “config:/path/to/item” and each “state:/path/to/item”, we will always receive the most recent value. Log compaction is not perfect in the sense that we will *only* receive the most recent value. We might receive a *few* older values before we reach the most recent one, but this still constitutes a vastly reduced message count compared to a complete, non-compacted message history.

KAFKA-BASED ALARM TOOLS

The alarm system tool implementation based on Kafka is significantly simpler than the previous implementation for the combination of RDB and JMS. We simply subscribe to Kafka messages and handle them as they are received. There is no need to buffer and then merge information from the RDB and JMS, i.e. from two different technologies.

Alarm Server

The alarm server typically runs as a Linux service. It subscribes to Kafka “config” messages, connects to the requested PVs and maintains their alarm state based on the same logic for latching and annunciating alarms as had been implemented in the original toolkit [1]. If offers a console for administrators to check on the alarm configuration and the state of PVs, but its main purpose is to send alarm “state” messages to clients.

Alarm User Interface

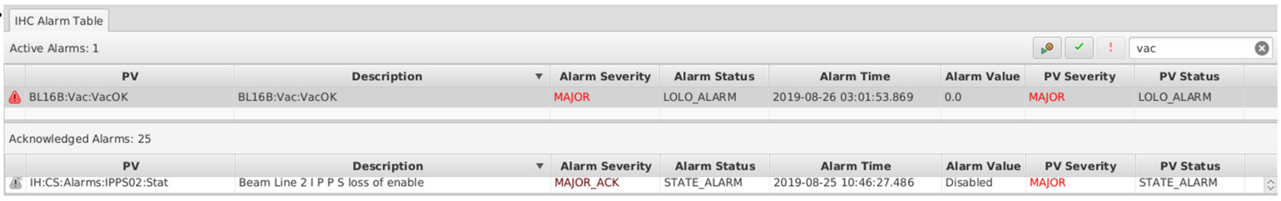
Alarm client tools subscribe to the “config” as well as “state” messages to obtain the alarm configuration and to display received alarms. Because of Log Compaction, a newly started alarm client may receive a small number of older alarms before it sees the most recent state, but the tools are capable of handling a high message rate, and the few additional messages are easily consumed.

The Alarm Table (Fig. 1) is the primary operator interface. It lists all active alarms. For the ideal case of a fault-free machine, it is empty, but as soon as alarms trigger, they are listed for operators to inspect. Operators can sort alarms by time, severity, PV name. They acknowledge them to indicate that the issue has been noted and is being handled. Acknowledged alarms move to the bottom section of the alarm table, which can be minimized to hide acknowledged alarms. Once the PV goes out of alarm state, it vanishes from the table.

The Alarm Tree display (Fig. 2) allows configuring the system by adding, removing and editing items. It can also be useful to monitor alarm states because it reflects the hierarchical configuration, indicating if several alarms are all clustered below one area of the configuration, or if they indicate a problem that, for example, affects several beam lines.

Both the Alarm Tree and Alarm Table offer access to alarm guidance information and related display links, but the Alarm Table only displays active alarms, while the Alarm Tree allows viewing and configuring all items, including those that are currently not in alarm.

Optional alarm system components that operators can start as desired include the Annunciator which performs audible annunciations of new alarms. The Area Panel displays colored blobs for the top-level elements of the alarm system configuration which can easily be recognized when viewed from a distance.


IHC Alarm Table

Active Alarms: 1

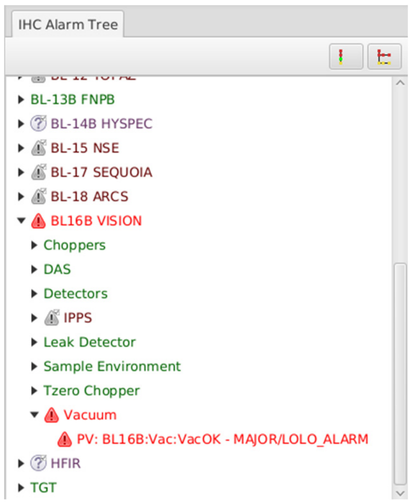
vac

PV	Description	Alarm Severity	Alarm Status	Alarm Time	Alarm Value	PV Severity	PV Status
BL16B:Vac:VacOK	BL16B:Vac:VacOK	MAJOR	LOLO_ALARM	2019-08-26 03:01:53.869	0.0	MAJOR	LOLO_ALARM

Acknowledged Alarms: 25

PV	Description	Alarm Severity	Alarm Status	Alarm Time	Alarm Value	PV Severity	PV Status
IH:CS:Alarms:IPPS02:Stat	Beam Line 2 I P P S loss of enable	MAJOR_ACK	STATE_ALARM	2019-08-25 10:46:27.486	Disabled	MAJOR	STATE_ALARM

Figure 1: Alarm table.


IHC Alarm Tree

- BL-13B FNPB
- BL-14B HYSPEC
- BL-15 NSE
- BL-17 SEQUOIA
- BL-18 ARCS
- BL16B VISION
  - Choppers
  - DAS
  - Detectors
  - IPPS
  - Leak Detector
  - Sample Environment
  - Tzero Chopper
  - Vacuum
    - PV: BL16B:Vac:VacOK - MAJOR/LOLO_ALARM
- HFIR
- TGT

Figure 2: Alarm tree.

Performance

The purpose of the alarm system is to assist operators, which limits the practical alarm rate to a level that human operators can handle [6]. The design of the tools is nevertheless such that it can handle a high number and rate of alarms.

Tests of the original implementation based on JMS and an RDB were able to import a configuration with 50000 PVs in about 5 minutes. The original alarm tree needed about 30 seconds to then load this configuration, and it could display about 10 alarm updates per second.

The new implementation based on Kafka can import 100000 PVs, i.e. a larger configuration, in only 10 seconds, which the alarm tree displays in about 10 seconds, then handling 500 state changes per second. More important than the vast improvement in importing a configuration is that fact that the new operator interface remains responsive. In the original implementation, the alarm tree was static until the complete configuration had been loaded. In the new implementation, the alarm tree keeps updating, displaying new items as they are loaded from the configuration. The new implementation does not need to distinguish between loading the configuration from an RDB and then updating it in response to JMS messages. Everything is based on handling a stream of Kafka messages, and loading the configuration is technically just an initial burst of messages that is handled the same way as any received Kafka message.

Observations

While the alarm system implementation based on Kafka is generally more performant than the previous RDB & JMS version, there is one peculiarity. The alarm system configuration is now a continuous stream of Kafka “config” messages. This works well for the distributed alarm server and client tools which are designed to handle online configuration changes, but it complicates the design of a tool for saving a configuration snapshot. There is technically no difference between the initial burst of “config” messages when loading the configuration, and a later “config” message sent when a user updates the configuration. To obtain a configuration snapshot, our current implementation reads configuration messages until no more changes are received for a certain time (4 seconds) and considers that the current configuration. If another configuration update is received while the tool writes the snapshot file, it issues a warning, so the user can decide to re-run the snapshot tool.

Upgrading

The new Kafka-based alarm system offers the same functionality as the previous version based on RDB and JMS, albeit with better performance. The alarm server logic for handling alarm state transitions is in fact exactly the same code, passing the same unit tests, thereby guaranteeing identical behaviour. However, the rest of the architecture is different. The new alarm server cannot interact with the older user interface tools and vice versa. To upgrade, both the CS-Studio user interface and the alarm server need to be updated at the same time. Both the old and the new tools allow importing and exporting the alarm configuration in the same XML file format. In practice it is therefore easy to export an existing configuration, update the user interface and alarm server to the current toolset, and then import the original alarm configuration.

ECOSYSTEM

By sending alarm messages via Kafka, additional tools can react to these messages. For example, an alarm logger service can forward messages to a generic message search and analysis tool like Elastic Search [7], which then allows alarm system maintainers to monitor, for example, the number of alarms over time, or to determine which alarm PVs trigger most frequently. Such information can be invaluable in detecting “noisy” alarms before they impact the productivity of operators who use the alarm system.

CONCLUSION

While the original alarm system toolkit has served several sites for about a decade, and continues to do so, the update from using a combination of RDB and JMS to Kafka allowed us to simplify the implementation and improve its performance headroom. Existing configurations can be transferred between the two implementations, providing a smooth upgrade path. The alarm system for the instrument hall of the Oak Ridge National Laboratory Spallation Neutron Source has successfully been using this new alarm system since January 2019.

REFERENCES

[1] K.-U. Kasemir, X. H. Chen, and E. Danilova, “The Best Ever Alarm System Toolkit”, in *Proc. 12th Int. Conf. on Accelerator and Large Experimental Physics Control Systems (ICALEPCS'09)*, Kobe, Japan, Oct. 2009, paper TUA001, pp. 46-48.

[2] L. R. Dalesio *et al.*, “EPICS 7 Provides Major Enhancements to the EPICS Toolkit”, in *Proc. 16th Int. Conf. on Accelerator*

and Large Experimental Physics Control Systems (ICALEPCS'17), Barcelona, Spain, Oct. 2017, pp. 22-26. doi:10.18429/JACoW-ICALEPCS2017-MOBPL01

[3] X. Geng, S. M. Hartman, and K.-U. Kasemir, "Alarm Rationalization: Practical Experience Rationalizing Alarm Configuration for an Accelerator Subsystem", in *Proc. 12th Int. Conf. on Accelerator and Large Experimental Physics Control Systems (ICALEPCS'09)*, Kobe, Japan, Oct. 2009, paper WEP109, pp. 606-608.

[4] K.-U. Kasemir, K. Shroff, "Future of CS-Studio", EPICS Meeting at *ICALEPCS2017*, https://indico.esss.lu.se/event/889/contributions/7050/

[5] Apache Kafka Documentation, Log Compaction, https://kafka.apache.org/documentation/#compaction

[6] B. Hollifield, E. Habibi, *The Alarm Management Handbook*, ISBN: 978-0-9778969-2-9, PAS, Inc. 2006.

[7] Elastic Search Documentation, https://www.elastic.co/guide