ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-THPDP068

IMPLEMENTING HIGH PERFORMANCE & HIGHLY RELIABLE TIME SERIES ACQUISITION SOFTWARE FOR THE CERN-WIDE ACCELERATOR DATA LOGGING SERVICE

M.W. Sobieszek, J. Woźniak, R. Mucha, P. Sowiński,
V. Baggiolini, C. Roderick, CERN, Geneva, Switzerland

Abstract

The CERN Accelerator Data Logging Service (NXCALS) stores data generated by the accelerator infrastructure and beam related devices. This amounts to 3.5TB of data per day, coming from more than 2.5 million signals from heterogeneous systems at various frequencies. Around 85% of this data is transmitted through the Controls Middleware (CMW) infrastructure. To reliably gather such volumes of data, the acquisition system must be highly available, resilient and robust. It also has to be highly efficient and easily scalable, as data rates and volumes increase, notably in view of the High Luminosity LHC.

This paper describes the NXCALS time series acquisition software, known as Data Sources. System architecture, design choices, and recovery solutions for various failure scenarios (e.g., network disruptions or cluster split-brain problems) are covered. Technical implementation details are discussed, covering the clustering of Akka Actors collecting data from tens of thousands of CMW devices. The NXCALS system has been operational since 2018 and has demonstrated the capability to fulfil all aforementioned requirements, while also ensuring self-healing capabilities and no data losses during redeployments.

INTRODUCTION

The CERN Accelerator Data Logging Service (NXCALS) stores data generated by the accelerator infrastructure and beam related devices, and in-turn makes this data available to the CERN community [1].

The NXCALS architecture is composed of three main subsystems (Figure 1): data acquisition and ingestion, data storage and compaction, and APIs or applications for data extraction. This paper focuses on the first subsystem, also known as NXCALS Data Sources (DS).

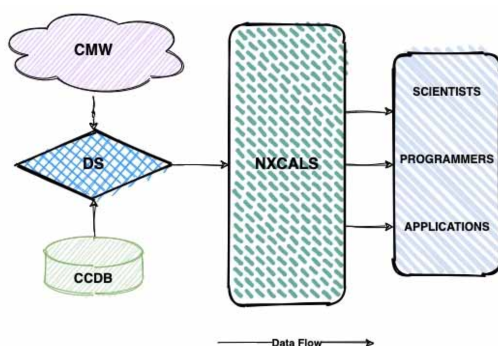

Figure 1: Overview of the NXCALS architecture.

The DS acquire data from the accelerator devices through the Controls Middleware (CMW) [2] using a subscription mechanism. One subscription corresponds to a data channel through which the device publishes data of one of its attributes (properties) to the DS. All information needed to configure the data sources and the subscriptions are managed within the central Controls Configuration Service (CCS) [3] and stored in the Controls Configuration Database (CCDB). Subscriptions are configured using a web interface (Controls Configuration Data Editor – CCDE), directly by end users (e.g. device experts, physicists, or operations teams) who know which data they want to be stored in NXCALS.

The NXCALS DS have to fulfil many requirements, but the most important is to reliably acquire huge amounts of data from accelerator infrastructure and beam related devices and transmit it to the NXCALS storage system without any data loss.

This is a surprisingly difficult requirement to fulfil. It needs a thorough analysis of the common challenges encountered when developing distributed systems.

The main concepts that must be addressed are:

- **Fault Tolerance and Resilience** - designing mechanisms to handle process failures (e.g., out of memory crashes) or network failures and partitions without service disruption.
- **Load Balancing** - distributing subscriptions evenly across processes to prevent overloading specific machines and their network interfaces.
- **Scalability** - being able to handle an increased number of subscriptions just by adding additional resources.
- **Security and Authentication** - implementing appropriate measures to protect data and prevent unauthorised access.
- **Monitoring and Debugging** - obtaining detailed information about the system performance and behaviour, especially when issues arise.

The rest of the paper will focus on how the NXCALS DS deal with the first three challenges in the list above.

SYSTEM OVERVIEW AND HIGH-LEVEL ARCHITECTURE

Figure 2 shows a high-level overview of the DS subsystem. Conceptually, the sequence of tasks executed by a data source process is straightforward:

1. The DS begins by retrieving subscriptions metadata from the configuration database.

© 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-THPDP068

2. Using CMW, the DS establishes a subscription to a device, from which it receives data.
3. If necessary, it applies basic data processing to the acquired data (e.g. filtering on change).
4. Finally, it forwards the processed data to the NXCALS ingestion subsystem.

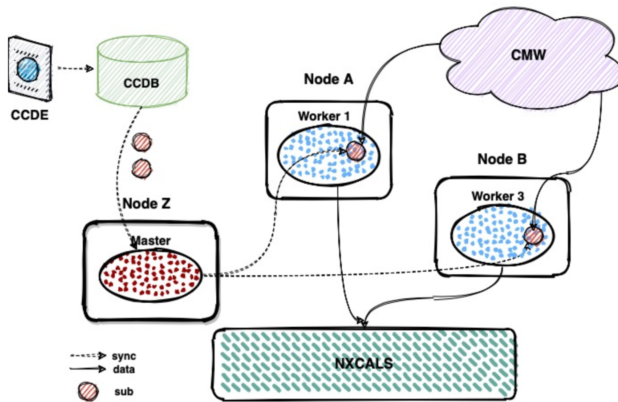

Figure 2: High-Level DS subsystem overview.

This workflow runs seamlessly for a relatively small number of subscriptions that can be managed by a single process. However, a single process is not enough to cope with the very high number of subscriptions and huge data flow present in the CERN accelerator complex. As such, the DS subsystem was built to distribute the workflow across multiple processes and machines, carefully coordinating all of them.

The Master-Worker model

A Master-Worker data processing model is used, which, as the name suggests, uses two primary types of actor processes:

1. Masters, that coordinate Workers.
2. Workers, that execute tasks, such as handling subscriptions and data.

A single Master instance assigns subscriptions to many Workers following a distribution strategy described later. The Workers handle the subscriptions assigned to them by subscribing to the relevant accelerator devices, processing incoming data, and forwarding it to the NXCALS system. In addition, there are Validators, a special kind of Worker, which checks whether a subscription complies with certain "validation criteria" regarding the data it processes (e.g. format and size). Validators are also orchestrated by the Master as described later. Master and Workers communicate as follows:

- At start-up, a Worker (or a Validator) registers with the Master and provides information about its processing capabilities.
- At runtime, Workers periodically communicate their status and various statistics about their subscriptions to their Master.
- The Master maintains a registry of available Workers and keeps track of their current workload and availability.

IMPLEMENTATION

Implementing the NXCALS DS subsystem was a challenging engineering task. Following a thorough analysis of the requirements and challenges mentioned in the Introduction, a suitable framework was selected to build upon, rather than re-implementing the wheel.

The Akka Framework

Akka [4] is a software framework that can be used to develop distributed, concurrent, fault-tolerant and scalable applications. By using the Actor Model, it raises the abstraction level and provides a better platform to build scalable, resilient, and responsive applications. Fault-tolerance is achieved by the "let it crash" paradigm that the telecom industry has used with great success to build self-healing applications and systems that never stop.

The structure of Akka systems is comprised of multiple Actors running in processes called Akka nodes, which are grouped together to form an Akka cluster. An Akka cluster typically runs on several physical machines.

Below is a list of Akka features, with a short explanation of why they were important for the NXCALS DS:

- **The Akka Actor Model** provides high-level abstraction for concurrent programming. Akka Actors handle state, ensure thread safety and make concurrency simpler. In the DS Actors are used to implement Masters, Workers and Validators.
- **Actor Location Transparency** abstracts away the physical location of Actors on machines, making it possible to interact with distributed Actors as if they were local. In the DS, this functionality allows to transparently use several physical machines and distribute the load over them.
- **Dynamic Membership** allows nodes to join and leave the cluster dynamically. This enables the DS cluster to easily scale out by adding machines as needed.
- **Supervision and Failure Detection** allows Actors to watch each other and detect failures. When one Actor watches another, the first Actor receives a notification when the second Actor terminates or becomes unavailable. This also works within an Akka Cluster and allows to detect when nodes become unreachable or crash. Supervision is very useful to achieve fault tolerance and resilience. A Master can supervise all its Workers and will be notified when any of them becomes unavailable. The Master can react appropriately by reassigning the failed Worker's subscriptions to other Workers.

In addition to implementing Masters, Workers and Validators using Akka Actors, NXCALS DS use Actors to implement most system components which run on separate Akka nodes located on different machines. These nodes take on specialized roles, such as *ValidationMaster*, *ValidationExecutor*, *SubscriptionManager*, and *WorkReceptionist*. Describing them goes beyond the scope of this paper.

Akka Actors communicate exclusively through asynchronous messages. By default, Akka implements at-most-

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-THPDP068

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

once delivery, meaning that each message sent by an Actor is delivered either zero or one times. In other words, messages may be lost [5], which is not acceptable for the NXCALS needs.

To address this issue, the communication protocol was modified to enable at-least-once delivery, thus ensuring that the receiver of a message will receive it. To do this, an Acknowledge and Retry mechanism was implemented. The Receiver must acknowledge each message from the Sender. If the Sender does not receive the acknowledgement within a specified time limit, it will retry by sending the initial message again.

MAIN WORKFLOWS

This section describes the main workflows that are executed in the Data Sources subsystem.

New Subscription

Depicted in Figure 3, this workflow starts when the Master detects that a new subscription has been configured in the CCDB. The Master tells the Validator to check if the new subscription complies with the validation criteria. If validation is successful, the Master assigns the subscription to a Worker. If not, the subscription is put into the Validation Waiting Room, where it waits to be re-validated after a configurable period.

Next, the Master uses a distribution strategy to select an appropriate Worker to handle the validated subscription. The default strategy is very straightforward and tries to ensure an even distribution of subscriptions among all Workers.

The Master typically assigns a new subscription to the Worker with the fewest subscriptions. It may also consider other criteria (so-called “labels” as described later). Finally, the Master tells the selected Worker(s) to start processing the data coming from the subscription.

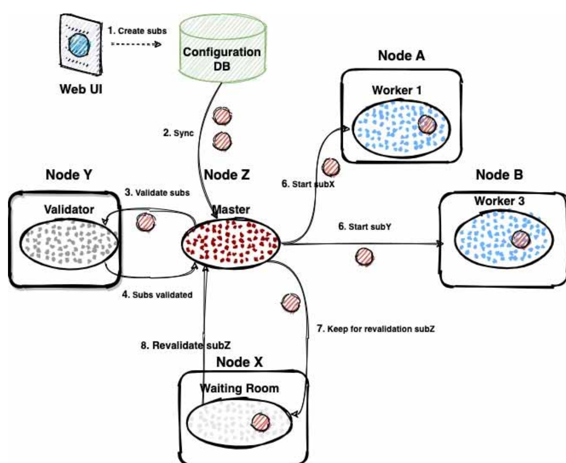

Figure 3: New subscription arriving in the system.

New Worker joining the cluster

When a Worker starts, it registers with the Master. If there are already existing Workers in the system with active subscriptions, the Master will promptly re-assign some subscriptions from the existing Workers to the new Worker (Figure 4). The goal is to evenly distribute subscriptions across all Workers.

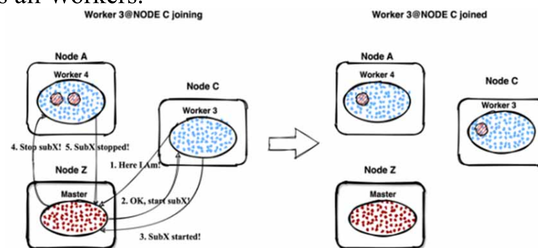

Figure 4: Worker joining the cluster.

Adding a subscription to an existing Worker

When the Master assigns a new subscription to a Worker, the following steps are executed (Figure 5):

1. The Worker activates the subscription it has been assigned.
2. It processes the resulting data flow by applying transformations defined at the subscription level. The transformations are being applied using several features provided by the Java RX library.
3. Finally, it forwards the transformed data to the NXCALS ingestion system.

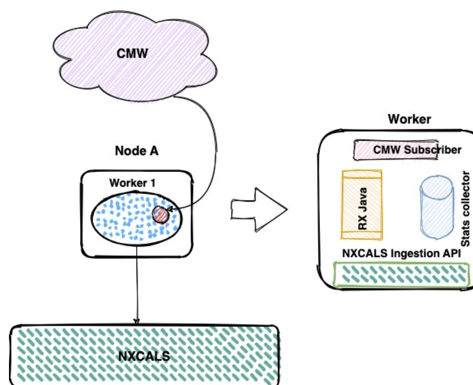

Figure 5: Worker operational mode.

Handling Subscription Failures

Workers monitor their active subscriptions and regularly report statistics about them to the Master. When a Worker notices that one of its subscriptions fails (disconnects), it informs the Master. The Master promptly tells the Worker to stop the subscription and hands it over to the Validator, which tries to re-establish and validate the subscription as described earlier.

Orderly Shutdown of a Worker

As mentioned, one of the crucial requirements of the system is to ensure no data losses. It strongly impacts the procedure for shutting down a Worker (e.g. for redeployment). Figure 6 illustrates the sequence of actions carried out during a standard Worker termination process:

1. When a Worker receives a signal to shut down, it sends a message to the Master to inform it.

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-THPDP068

2. The Master acknowledges the Worker's message and redistributes all its subscriptions to other available Workers.
3. Once the Master receives confirmation that all the Worker's subscriptions have been initiated elsewhere, it instructs the terminating Worker to stop all its subscriptions.
4. Following this, the Worker proceeds with its own termination.

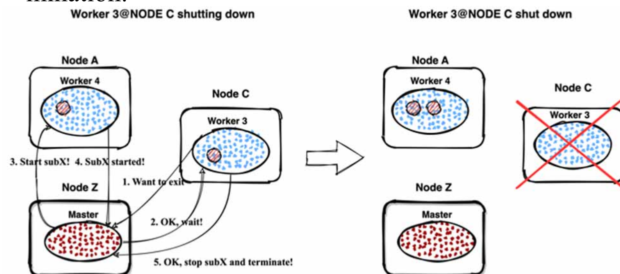

Figure 6: Worker leaving the cluster.

During this process, it may happen that a given subscriptions is active on two different Workers (the terminating one and the one taking over). This will result in duplicate data being sent to NXCALS during this time. This is not a problem, because the NXCALS ingestion system deduplicates data during a nightly data merging and compaction process.

FURTHER IMPLEMENTATION DETAILS

State Persistence

The Master keeps track of available Workers and their current subscription assignments. It persists this state in external storage using the Akka Persistence extension. This extension provides storage and retrieval of domain events to maintain the system's state and preserves it across system restarts. Concretely, for the NXCALS DS, the akka-persistence-jdbc extension is used to store the state in an Oracle database.

The Master persists snapshots of subscription assignments whenever they change, which means that the persisted state is always up-to-date. When a new Master starts, it simply retrieves the most recent saved snapshot by its predecessor and becomes fully operational. No state needs to be persisted for the Workers or the Validator.

Achieving Fault Tolerance and Resilience

The NXCALS DS subsystem must be highly available and resilient, even in the case of node crashes and network problems.

Unfortunately, it is impossible to distinguish between node crashes and temporary network failures. The Akka cluster does have a failure detector that will notice node crashes and network partitions, but it cannot differentiate between the two failure cases.

The naive approach to handle any failure would be to remove an unreachable node from the cluster after a timeout. This works fine for crashes and short, transient network partitions, but it does not work for longer network outages. A longer network failure creates two separated groups of nodes, whereby each group sees the other one as crashed or unreachable. This may eventually lead to creation of two separated, disconnected clusters that each continue to work on their own. This problem is known as a "Split-Brain scenario" and may lead to inconsistent behaviour or data corruption. Below, the different crash scenarios are described, together with how they are handled.

Handling a Worker Crash

As described previously, the Master monitors the availability of all active Worker nodes, using the Akka supervision mechanism. It receives notifications when a Worker crashes or becomes unavailable due to network problems. The Master then reassigns all its subscriptions to other Workers that are currently operational and capable of processing the subscriptions.

Handling a Master Crash

In the NXCALS DS, the Cluster Singleton pattern is used to react to a crash of the Master. This pattern ensures that only one instance of a specific type of Actor (in this case, one instance of the Master) is active within the cluster at any given time. In the current production setup, three Master instances exist, but only one is active at a given time. If the active Master goes down (whether due to redeployment or a crash), one of the other two, typically the oldest one, takes over. The new Master retrieves its state from the snapshots that the previous Master persisted, and then retrieves data from the configuration database to incorporate any potential changes made in the meantime.

Handling a Split-Brain Problem

Handling a Split-Brain scenario is more challenging. Among the strategies Akka provides for dealing with this problem, the *Static Quorum* was chosen.

In essence, this strategy involves downing the unreachable nodes if the number of remaining healthy nodes is equal or greater to a predefined constant known as the quorum size. Simply put, the quorum size defines the minimal number of nodes required for the cluster to remain operational.

This quorum typically determines the number of the members in the cluster and should not be greater than *quorum-size*2 - 1*. For the NXCALS DS, the quorum size is based on the number of Master instances. For 3 Master instances, the quorum size is 2, and this number is established and fixed during the initial configuration of the cluster.

It is important to highlight that in the cluster that is shut down, the Master instances will exit automatically because they are subject to the Static Quorum mechanism. At the same time, the Workers and their subscriptions continue to run, which avoids any data loss. Naturally, the active Master in the healthy, surviving cluster will consider these subscriptions as missing, and will re-create and redistribute them to its own Workers. This approach allows to avoid any data loss that may otherwise occur during the time needed to detect an unhealthy Worker (and its otherwise self-downing procedure) by the Master and the subsequent redistribution of their subscriptions.

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-THPDP068

Achieving Scalability

It is important to underline again that the Workers self-register to the Master, rather than being spawned by it. This design enhances the system's scalability and makes it possible to dynamically increase processing capacity, simply by adding new Workers, that will seamlessly join the system, and process subscriptions allocated to them by the Master (Fig. 7).

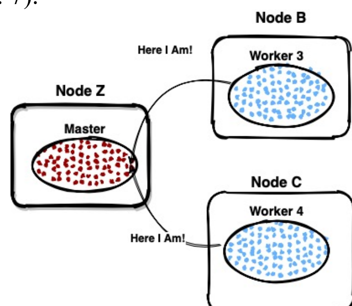

Figure 7: Worker registering to Master.

Subscription Labelling

Subscriptions and the data they produce have different characteristics. For instance, some subscriptions can produce heavy/big data with small frequencies, while others can produce small data with higher frequencies, or a combination of "heavy" data published at high frequencies.

Labels have been introduced to the NXCALS DS which can be used to mark subscriptions according to their characteristics. These labels are simply a property of a subscription and can be customized by users or system maintainers. For example, mission critical subscriptions can be labelled as such.

The labels are used by the Master when distributing subscriptions to Workers. For example, it can assign heavy subscriptions or mission-critical subscriptions to dedicated Workers or nodes.

DEPLOYMENT EXPERIENCE

The NXCALS DS subsystem is deployed in production since 2019 and has demonstrated remarkable robustness and high availability. The DS currently handle around 90k subscriptions, which produce on average, 100k messages a second, and 3.5TB of data a day.

On several occasions, the DS have been exposed to subscriptions that, due to misconfiguration by equipment experts, have produced abusive amounts of data. The DS survived these events, thanks to the labelling mechanism, that allowed the bad subscriptions to be isolated, and therefore avoid any data losses on the other subscriptions.

It was also observed on several occasions (mainly during infrastructure maintenance periods) how the Akka cluster shuts itself down due to network problems and then fully recreates itself after the network is restored, without any data losses.

SUMMARY

This paper has given an overview of the requirements for the NXCALS Data Sources, with the overarching imperative of not losing any data during acquisition and ingestion.

The chosen architecture has been presented, including the reasoning for implementing the Data Sources as a distributed system, using a Master/Worker paradigm, and the engineering challenges related to this. The Akka platform has been described, together with an explanation of how its features serve as a strong basis for the implementation of the NXCALS Data Sources.

The resulting system has demonstrated remarkable robustness and high availability, and there is confidence that it will scale with the ever-growing data volumes that will be produced by the upcoming LHC High Luminosity project.

The use of the Akka Platform and the modular design of the Data Sources described in this paper are not specific to CERN. Therefore, this software could be adapted to meet similar data acquisition needs in other institutes and domains.

REFERENCES

[1] J. Woźniak and C. Roderick, "NXCALS – Architecture and challenges of the next CERN accelerator logging service", in *Proc. 17th Int. Con. on Accelerator and Large Experimental Physics Control Systems (ICALEPCS'19)*, New York, USA, Oct. 2019, pp. 1465-1469. doi:10.18429/JACoW-ICALEPCS2019-WEPHA163

[2] W. Sliwinski and J. Lauener, "How to design & implement a modern communication middleware based on ZeroMQ", in *Proc. 16th Int. Con. on Accelerator and Large Experimental Physics Control Systems (ICALEPCS'17)*, Barcelona, Spain, Oct. 2017, pp. 45-51. doi:10.18429/JACoW-ICALEPCS2017-MOBPL05

[3] L. Burdzanowski *et al.*, "CERN Controls Configuration Service - a challenge in usability", in *Proc. 16th Int. Con. on Accelerator and Large Experimental Physics Control Systems (ICALEPCS'17)*, Barcelona, Spain, Oct 2017, pp 159-165. doi:10.18429/JACoW-ICALEPCS2017-TUBP

[4] https://akka.io

[5] https://www.infoq.com/articles/no-reliable-messaging/

© 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